

MULTI: Debugging



Green Hills Software, Inc.

**30 West Sola Street
Santa Barbara, California 93101
USA**

Tel: 805-965-6044

Fax: 805-965-6343

www.ghs.com

DISCLAIMER

GREEN HILLS SOFTWARE, INC. MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. Further, Green Hills Software, Inc. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Green Hills Software, Inc. to notify any person of such revision or changes.

Copyright © 1983-2009 by Green Hills Software, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission from Green Hills Software, Inc.

Green Hills, the Green Hills logo, CodeBalance, GMART, GSTART, INTEGRITY, MULTI, and Slingshot are registered trademarks of Green Hills Software, Inc. AdaMULTI, Built with INTEGRITY, EventAnalyzer, G-Cover, GHnet, GHnetLite, Green Hills Probe, Integrate, ISIM, u-velOSity, PathAnalyzer, Quick Start, ResourceAnalyzer, Safety Critical Products, SuperTrace Probe, TimeMachine, TotalDeveloper, DoubleCheck, and velOSity are trademarks of Green Hills Software, Inc.

All other company, product, or service names mentioned in this book may be trademarks or service marks of their respective owners.

PubID: 10686
August 25, 2014

Contents

	Preface	xv
	About This Book	xvi
	The MULTI 5 Document Set	xvii
	Conventions Used in the MULTI Document Set	xviii
1	Introduction	1
	The MULTI Integrated Development Environment	2
	MULTI Launcher	2
	Editing Tools	2
	Building Tools	2
	Debugging Tools	3
	Administrative Tools	3
	The MULTI Launcher	4
	MULTI Workspaces	5
	The MULTI Debugger	6
	Accessing Online Help	8
	Full Online Manuals	8
	Context-Sensitive Help	8
	Command and Configuration Option Help	8
	The Help Viewer Window	9
	Navigating Manuals in the Help Viewer	10
Part I	Basic Debugging	
2	Before You Begin Debugging	17
	Generating Debugging Information	18
	Programs Compiled with Green Hills Compilers	18
	Starting the MULTI Debugger	20
	Starting in Graphical Mode	20
	Starting in Non-Graphical Mode (UNIX only)	21
	Using the Command Line	22
	Next Steps	27

Contents

3	The Main Debugger Window	29
	Debugger Window Components	30
	Setting Up Your Debugging Environment	32
	Reusing the Debugger Window	32
	Opening Multiple Debugger Windows	32
	The Target List	33
	Repositioning and Hiding the Target List	33
	Debugging Target List Items	34
	Target Terminology	36
	The Status Column	37
	The CPU % Column	38
	The Source Pane	39
	Source Pane Display Modes	41
	Source Pane Line Numbers	42
	The Navigation Bar	43
	The Cmd, Trg, I/O, Srl, Py, and Tfc Panes	44
	The Command Pane	45
	The Target Pane	48
	The I/O Pane	48
	The Serial Terminal Pane	49
	The Python Pane	50
	The Traffic Pane	50
	The Status Bar	53
4	Connecting to Your Target	55
	Working with Connection Methods	56
	Tools Overview	57
	Standard Connection Methods	57
	Creating a Standard Connection Using the Project Wizard	57
	Creating a Standard Connection Using the Connection Chooser	58
	Configuring a Standard Connection with the Connection Editor	59
	Connecting with a Standard Connection	66
	Custom Connection Methods	67
	Temporary Connection Methods	69
	Simulator Connections	70

Contents

Native Development Connections	70
Freeze-Mode and Run-Mode Connections to the Same Target	70
Automatically Established Run-Mode Connections	71
Troubleshooting	74
Using the Connection Organizer	75
Opening the Connection Organizer	76
Creating a Connection Method	77
Editing a Connection Method	77
Creating and Managing Connection Files	78
Connecting from the Connection Organizer	79
Managing Your Connected Targets	80
Connection Organizer Menu and Action Reference	80
Disconnecting from Your Target	84
5 Preparing Your Target	85
Chapter Terminology	86
Associating Your Executable with a Connection	87
Updating Your MULTI 4 Target Connections	88
Preparing Your Target	90
The Prepare Target Dialog Box	91
Program Types	92
Downloading Your Executable	95
Verifying the Presence of Your Executable	95
Related Settings	97
Memory Sensitive Mode	97
No Stack Trace Mode	98
6 Executing and Controlling Your Program from the Debugger	99
Starting and Stopping a Program	100
Single-Stepping Through a Program	101
Using Breakpoints and Tracepoints	102
Breakdots, Breakpoint Markers, and Tracepoint Markers	103
Viewing Breakpoint and Tracepoint Information	105
Working with Software Breakpoints	105
Working with Hardware Breakpoints	110

Contents

	Working with Shared Object Breakpoints	113
	Advanced Breakpoint Topics	114
	Software and Hardware Breakpoint Editors	115
	The Breakpoints Window	123
	Breakpoints Window Columns	123
	Breakpoints Window Buttons	125
	Breakpoints Window Mouse and Keyboard Actions	126
	Breakpoints Window Shortcut Menu	127
7	Navigating Windows and Viewing Information	129
	Navigating and Searching Basics	130
	Using the Scroll Bar	130
	Incremental Searching	130
	Searching in Files	133
	Selecting, Cutting, and Pasting Text	134
	Navigating, Browsing, and Searching the Source Pane	136
	Using the File Locator	139
	Using the Procedure Locator	140
	Using Navigation History Buttons	142
	Using the Source Pane Search Dialog Box	143
	Viewing Program and Target Information	144
	Viewing Information in Stand-Alone Windows	144
	Viewing Information in the Command Pane	150
8	Using Debugger Notes	153
	Creating and Editing Debugger Notes	154
	Editing a Single Debugger Note	154
	Editing Multiple Debugger Notes	156
	Removing Debugger Notes	156
	Organizing Debugger Notes Into Groups	157
	Viewing Debugger Notes	157
	The Note Browser	158

Contents

Part II Viewing Debugging Information and Program Details

9	Viewing and Modifying Variables with the Data Explorer	163
	Opening a Data Explorer	164
	The Data Explorer Window	165
	The Data Explorer Toolbar	166
	The Edit Bar	167
	Viewing Multiple Items in a Data Explorer	168
	Updating Data Explorer Variables	169
	Freezing Data Explorer Variables	169
	Types of Variable Displays in a Data Explorer	170
	Displaying Structures	170
	Displaying Linked Lists	171
	Displaying Arrays	173
	Displaying C++ Classes	174
	Changing How Variables are Displayed in a Data Explorer	175
	Changing the Type Used to Display a Variable	175
	Viewing Pointers to C++ Base Classes	175
	Modifying Variables from a Data Explorer	177
	Configuring Data Explorers	178
	Configuring Individual Data Explorer Dimensions	178
	Setting Global Options for Data Explorers	178
	Data Explorer Messages	181
	Data Explorer Menus	182
	The Edit Menu	182
	The View Menu	183
	The Format Menu	184
	The Evaluate Menu	187
	The Tools Menu	188
	The Settings Menu	189
10	Browsing Program Elements	191
	Browsing Program Elements Overview	192
	The Browse Window	194

Contents

Browse Window Basics	194
Filtering Content in the Browse Window	196
Browse Window Headings	200
Browsing Procedures	200
Browsing Global Variables	209
Browsing Source Files	211
Browsing Data Types	214
Browsing Cross References	216
The Tree Browser Window	220
Opening a Tree Browser	220
Using a Tree Browser	220
Browsing Classes	224
Browsing Static Calls By Function	225
Browsing Static Calls By File	226
Browsing Dynamic Calls by Function	226
The Graph View Window	227
Browsing Includes	228
Controlling Layout and Navigating in Graph View	228
11 Using the Register Explorer	233
The Register View Window	234
The Menu Bar	236
Toolbar	239
Tabs	240
Register Tree	241
Performing Actions on Registers Using the Shortcut	
Menus	242
Customizing the Register View Window	243
Copying Register Values	247
Editing Register Contents	247
Controlling Refresh of Register Values	248
Printing the Window Contents	249
Register View Window Configuration Files	249
The Register Information Window	250
Concise Display Pane	252
Detailed Display Pane	253
Help Pane	254
Button Set	255
Changing a Register Value	255
The Modify Register Definition Dialog	256

Contents

Bitfield Editor Pane	257
Bitfield List Pane	258
The Register Setup Dialog	259
The Register Search Window	260
Using Debugger Commands to Work with Registers	262
Customizing Registers	265
Customizing Registers from the Command Line	265
Customizing Registers in Default .rc Files	265
Customizing Default Register Definition Files	265
12 Using Expressions, Variables, and Procedure Calls	271
Evaluating Expressions	272
Expression Scope	272
Expressing Source Addresses	273
Language-Independent Expressions	274
Language Keywords	274
Caveats for Expressions	274
Viewing Variables	277
Variable Lifetimes	278
Examining Data	279
Variables	279
Examining Preceding Memory Locations	280
Expression Formats	280
Language Dependencies	284
Wildcards	284
Procedure Calls	285
Caveats for Command Line Procedure Calls	286
Macros	288
MULTI Special Variables and Operators	290
User-Defined Variables	290
System Variables	291
Processor-Specific Variables	298
Special Operators	298
Syntax Checking	301

Contents

13	Using the Memory View Window	303
	Setting the Active Location	305
	Locking the Current Symbol's Location	305
	The Memory Pane	306
	Formatting View Columns	306
	Managing View Columns	307
	Controlling Memory Access	308
	Freezing the Memory View Window	308
	Setting Access Sizes	308
	Disabling Block Reads	309
	Editing Memory	309
	The Memory View Toolbar	310
	Memory View Menus	312
	The Memory Menu	312
	The View Menu	314
	The Shortcut Menu	316
14	Viewing Memory Allocation Information	319
	Debugging Memory Allocation Errors	320
	General Memory Allocation Checking	320
	Intensive Memory Allocation Checking	321
	Using Memory Allocation Library Functions	322
	Using the Memory Allocations Window	325
	Viewing the Memory Allocation Visualization	328
	Viewing Memory Leaks and Allocation Information	332
15	Collecting and Viewing Profiling Data	339
	Types of Profiling Data	340
	Overview of Profiling Methods	341
	Generating Profiling Data for a Task, AddressSpace, or Stand-Alone Program	342
	Collecting Profiling Data	343
	Collecting Profiling Data for All Tasks in Your System	344
	Collecting Profiling Data for a Task, AddressSpace, or Stand-Alone Program	345

Contents

	Caveats for Profiling	346
	Disabling the Collection of Profiling Data	347
	Viewing Profiling Information	347
	The Profile Window	348
	Continual Updates in the Profile Window	349
	Profiling Reports	350
	The Profile Window Reference	357
	Viewing Profiling Information in the Debugger	363
	Performing Range Analyses	364
	Managing Profiling Data	366
	Adding to or Overwriting Existing Profiling Data	366
	Manually Dumping Profiling Data	367
	Manually Processing Profiling Data	368
	Clearing Profiling Data	370
	Caveats for Dumping and Clearing Profiling Data	370
	Using the protrans Utility	371
16	Using Other View Windows	375
	Viewing Call Stacks	376
	The Call Stack Window and Command Line	
	Procedure Calls	377
	The Call Stack Window and Procedure Prologues and Epilogues	377
	Viewing Native Processes	378
	Viewing Caches	379
	The Cache View Window	379
	The Cache Find Window	383
Part III	Advanced Debugging in Specific Environments	
17	Testing Target Memory	389
	Quick Memory Testing: Using the Memory Test Wizard	390
	Advanced Memory Testing: Using the Perform Memory Test Window	394
	Defining Memory Areas to Test	395
	Selecting Memory Tests	397

Contents

Setting Test Options	399
Specifying Test Methods	400
Running Memory Tests	401
Viewing Memory Test Results	402
Continuous Memory Testing	403
Memory Testing with a Target Agent	403
Types of Memory Tests	404
Address Bus Walking Test	404
Data Bus Walking Test	406
Data Pattern Test	408
Memory Read Test	411
CRC Compute	411
CRC Compare	412
Find Start/End Ranges	412
Efficient Testing Methods	414
Running Memory Tests from the Command Line	414
Detecting Coherency Errors	417
Checking Coherency Manually	418
Checking Coherency Automatically	418
18 Run-Mode Debugging	421
Establishing Run-Mode Connections	422
Loading a Run-Mode Program	423
The Task Manager	424
Working with AddressSpaces and Tasks	425
Run-Mode AddressSpaces	425
Attaching to Tasks	425
Halting Tasks On Attach	426
Debugging Run-Mode Tasks	426
Freezing the Task Manager's Task List Display	427
Working with Task Groups in the Task Manager	428
Default Task Groups	428
Configuring Task Group Settings	430
Working with Run-Mode Breakpoints	430
Task-Specific Breakpoints	430
Any-Task Breakpoints	430
Group Breakpoints	431
Synchronous Operations	432

Contents

Task Manager Options	435
Menu Bar	435
Toolbar	439
Task List Pane	440
Information Pane	441
The Task Manager Shortcut Menu	442
Task Group Configuration File	444
OS-Awareness in Run Mode	445
The OSA Explorer	445
The OSA Object Viewer	446
Profiling All Tasks in Your System	447
19 Freeze-Mode Debugging and OS-Awareness	449
Starting MULTI for Freeze-Mode Debugging and OS-Awareness	451
The OSA Explorer	452
Displaying an OSA Explorer	452
The OSA Explorer GUI Reference	455
Customizing the Object List	456
Object List Operations	456
Debugging in Freeze Mode	457
Master Debugger Mode	458
Task Debugger Mode	459
Working with Freeze-Mode Breakpoints	462
Program I/O	463
Multi-Core Debugging	464
Configuration File	468
General Settings	470
The Object Type Definition	473
The Object Definition	473
Example: Configuration File	475
20 Establishing Serial Connections	481
Starting the Serial Terminal Emulator (MTerminal)	482
Using Quick Connect	483
Creating and Configuring Serial Connections	484
Using the Serial Connection Chooser	484

Contents

Using the Serial Connection Settings Dialog	485
The MTerminal Window	488
The MTerminal Menu Bar	488
The MTerminal Toolbar	490
The MTerminal Status Bar	491
Running the Serial Terminal Emulator in Console Mode	492
mterminal Syntax and Arguments	492

Part IV Appendices

A	Debugger GUI Reference	497
	Debugger Window Menus	498
	The File Menu	499
	The Debug Menu	503
	The View Menu	512
	The Browse Menu	518
	The Target Menu	519
	The Tools Menu	523
	The Config Menu	525
	The Windows Menu	530
	The Help Menu	530
	The Debugger Window Toolbar	531
	Configuring the Debugger Toolbar	538
	The Target List Shortcut Menu	542
	Source Pane Shortcut Menus	543
	Common Shortcut Menu Options	545
	The Source Line Shortcut Menu	545
	The Procedure Shortcut Menu	548
	The Variable Shortcut Menu	549
	The Type Shortcut Menu	550
	The Type Member Shortcut Menu	551
	The C/C++ Macro Shortcut Menu	551
	The Command Pane Shortcut Menu	552
	The Source Pane Search Dialog Box	552
	The File Chooser Dialog Box (UNIX)	554
B	Keyboard Shortcut Reference	557
	Main Debugger Window Shortcuts	558

Contents

	Source Pane Shortcuts	560
	Command Pane Shortcuts	561
C	Command Line Reference	563
D	Creating Custom Data Visualizations	571
	Using MULTI Data Visualization (.mdv) Files	574
	Loading and Clearing MULTI Data Visualization (.mdv) Files	576
	Invoking Customized Data Visualizations	577
	MULTI Data Visualization (.mdv) File Format	578
	Data Descriptions	578
	Profile Descriptions	600
	View Descriptions	602
	Using Expressions in MULTI Data Visualization (.mdv) Files	604
	.mdv File Examples	605
E	Register Definition and Configuration Reference	609
	The GRD Register Definition Format	610
	The general Section	611
	The enum Section	618
	The structure Section	621
	The bitfield Section	623
	The register Section	626
	The group Section	632
	Register View Window Configuration Files	634
	Default Loading of Configuration Files	635
	File Syntax	635
	File Header	635
	Describing Tabs	636
	Index	639

Contents

Preface

This Preface Contains:

- About This Book
- The MULTI 5 Document Set
- Conventions Used in the MULTI Document Set

This section discusses the purpose of the manual, typographical conventions used, and the MULTI documentation set.

About This Book

The *MULTI: Debugging* book provides comprehensive documentation of the features of the MULTI Debugger. It is divided into the following parts:

- *Part I: Basic Debugging* documents the basic features of the Debugger. It also describes how to create and collect debugging information, how to connect to and prepare your target for debugging, how to navigate the Debugger window, and how to execute embedded programs and control and monitor embedded processes from the Debugger. See page 15.
- *Part II: Viewing Debugging Information and Program Details* documents how to examine all aspects and elements of your program code and how to view memory allocation, profiling, and other debugging information. See page 161.
- *Part IV: Advanced Debugging in Specific Environments* documents how to test target memory; how to use flash memory; how to use the MULTI Debugger with ROM; how to perform field debugging, run-mode debugging, freeze-mode debugging, and OS-aware debugging; and how to establish serial connections. See page 387.
- *Part V: Appendices* contains reference material, including comprehensive documentation of the Debugger's default menu items, shortcut menus, keyboard shortcuts, and command line options. This part also contains instructions about how to use the Debugger with third-party tools and how to create custom data visualizations. Specifications for the register definition format supported by this release are also included. See page 495.



Note New or updated information may have become available while this book was in production. For additional material that was not available at press time, or for revisions that may have become necessary since this book was printed, please check your MULTI installation directory for release notes, **README** files, and other supplementary documentation.

The MULTI 5 Document Set

The primary documentation for using MULTI is provided in the following books:

- *MULTI: Getting Started* — Describes how to install MULTI, and takes you through creating, building, and debugging an example project.
- *MULTI: Licensing* — Describes how to obtain, install, and administer Green Hills licenses. This book is intended for system administrators.
- *MULTI: Editing Files and Configuring the IDE* — Describes how to use the MULTI Editor and MULTI Launcher, how to use a version control system with MULTI, and how to configure the MULTI IDE.
- *MULTI: Building Applications* — Describes how to use the MULTI Project Manager and compiler drivers and the tools that compile, assemble, and link your code. Also describes the Green Hills implementation of supported high-level languages.
- *MULTI: Configuring Connections* — Describes how to set up your target debugging interface for use with MULTI and how to configure connections to your target.
- *MULTI: Debugging* — Describes how to use the MULTI Debugger and its associated tools.
- *MULTI: Debugging Command Reference* — Describes how to use Debugger commands and provides a comprehensive reference of Debugger commands.

For a comprehensive list of the books provided with your MULTI installation, see the **toc_target.pdf** file located in the **manuals** subdirectory of your installation.

Most books are available in the following formats:

- A printed book (select books are not available in print).
- Online help, accessible from most MULTI windows via the **Help** → **Manuals** menu.
- An electronic PDF, available in the **manuals** subdirectory of your installation.

Conventions Used in the MULTI Document Set

All Green Hills documentation assumes that you have a working knowledge of your host operating system and its conventions, including its command line and graphical user interface (GUI) modes. For example, you should know how to use basic commands, how to open, save, and close files, and how to use a mouse and standard menus.

Green Hills documentation uses a variety of notational conventions to present information and describe procedures. These conventions are described below.

Convention	Meaning	Example
bold type	Indicates a: • Filename or pathname • Command • Option • Window title • Menu name or menu choice • Field name • Button name	• C:\MyProjects • setup command • -G option • The Breakpoints window • The File menu • Working Directory: • The Browse button
<i>italic type</i>	Indicates: • Replaceable text • A new term • A book title	• -o <i>filename</i> • A task may be called a <i>process</i> or a <i>thread</i> • <i>MULTI: Debugging</i>

Convention	Meaning	Example
monospace type	Indicates: • Text you should enter as presented • A word or words used in a command or example • Source code • Input/output • A function	• Type <code>help</code> <code>command_name</code> • The wait [-global] command blocks command processing, where -global blocks command processing for all MULTI processes. • <code>int a = 3;</code> • <code>> print Test</code> <code>Test</code> • <code>GHS_System()</code>
ellipsis (...) (in command line instructions)	Indicates that the preceding argument or option can be repeated zero or more times.	debugbutton [name]...
greater than sign (>)	Represents a prompt. Your actual prompt may be a different symbol or string. The > prompt helps to distinguish input from output in examples of screen displays.	<code>> print Test</code> <code>Test</code>
pipe () (in command line instructions)	Indicates that one (and only one) of the parameters or options separated by the pipe or pipes should be specified.	call <i>proc</i> <i>expr</i>
square brackets ([]) (in command line instructions)	Indicate optional arguments, commands, options, and so on. You can either include or omit the enclosed elements. The square brackets should not appear in your actual command.	<code>.macro</code> <i>name</i> [<i>list</i>]

The following command description demonstrates the use of some of these typographical conventions.

gxyz [-option]... *filename*

The formatting of this command indicates that:

- The command **gxyz** should be entered as shown.
- The option *-option* should either be replaced with one or more appropriate options or be omitted.
- The word *filename* should be replaced with the actual filename of an appropriate file.

The square brackets and the ellipsis should not appear in the actual command you enter.

Introduction

Chapter 1

This Chapter Contains:

- The MULTI Integrated Development Environment
- The MULTI Launcher
- The MULTI Debugger
- Accessing Online Help

This chapter provides an overview of the MULTI Integrated Development Environment.

The MULTI Integrated Development Environment

MULTI is a complete Integrated Development Environment (IDE) designed especially for embedded systems engineers to assist them in compiling, optimizing, debugging, and editing embedded applications.

The MULTI IDE includes the following tools for each part of the software development process. Many of the MULTI tools can be launched from within the IDE or as separate stand-alone programs. Parenthetical text indicates corresponding executable files, which are located in your MULTI installation.

MULTI Launcher

MULTI Launcher (mstart) — The gateway to the MULTI IDE, which allows you to quickly launch any of the primary MULTI tools, access open windows, and manage MULTI workspaces.

Editing Tools

- **MULTI Editor (me)** — A graphical editor for modifying text files.
- **Checkout Browser (mcobrowse)** — A graphical viewer for files managed under a version control system.
- **Diff Viewer (diffview)** — A graphical viewer that displays differences between two text files.

Building Tools

- **MULTI Project Manager (mprojmgr)** — A graphical interface for managing and building projects.
- **Integrate (integrate)** — A graphical utility for configuring tasks, connections, and kernel objects across multiple address spaces when using the INTEGRITY RTOS.

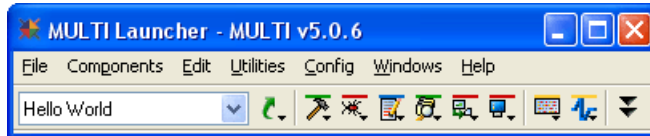
Debugging Tools

- **MULTI Debugger (multi)** — A graphical source-level debugger.
- **Instruction Set Simulator** — A tool that simulates your embedded processor on your development host.
- **EventAnalyzer (mevgui)** — A graphical viewer for monitoring the complex real-time interactions of an embedded RTOS such as INTEGRITY or u-velOSity.
- **ResourceAnalyzer (wperf)** — A graphical viewer for monitoring the CPU and memory usage of an embedded system running the INTEGRITY RTOS.
- **Serial Terminal (mterminal)** — A serial terminal emulator for connecting to serial ports on embedded devices.
- **DoubleCheck** — A static source code analysis tool for locating programming problems that lead to security and reliability failures in software.
- **MULTI TimeMachine Tool Suite** — A wide variety of trace analysis tools that dramatically extend MULTI's trace and debugging capabilities.

Administrative Tools

- **Bug Report (gbugrpt)** — A utility for providing system configuration and tool version information to the Green Hills support staff.
- **Green Hills Probe Administrator (gadmin)** — A graphical interface for configuring and managing Green Hills Probes and SuperTrace Probes.
- **Graphical Utilities (wgutils)** — A collection of utilities for analyzing and performing various operations on object files, libraries, and executables produced with the Green Hills toolchain.
- **MULTI License Administrator (mlmadmin)** — A graphical utility for managing Green Hills tools licenses.

The MULTI Launcher



The MULTI Launcher (**mstart**) provides a convenient way to launch frequently used tools, to create new or access recently used files and projects, and to manage MULTI workspaces. All of the main MULTI components can be accessed using the following buttons:

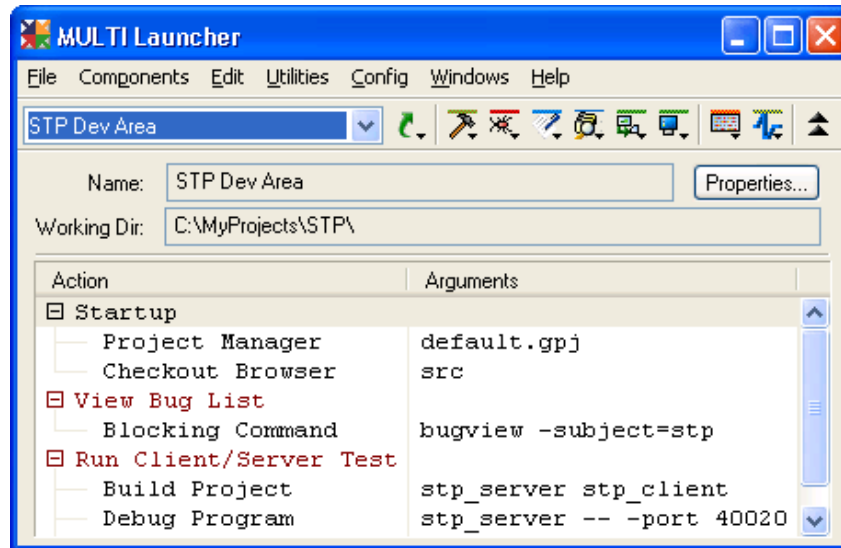
-  — Runs a shortcut or an action sequence in the current workspace. Also allows you to create a new workspace or create or edit a shortcut.
-  — Opens the Project Manager on a recent or new project.
-  — Opens the Debugger on a recent or new executable.
-  — Opens the Editor on a recent or new file.
-  — Opens the Checkout Browser on a recent or new checkout.
-  — Establishes a target connection or opens the Connection Chooser or Connection Organizer.
-  — Opens a Serial Terminal using a recent or new connection.
-  — Opens the EventAnalyzer (licensed separately).
-  — Opens the ResourceAnalyzer (licensed separately).
-  — Shows/hides the detail pane of the Launcher.

You can also launch the Green Hills License Administrator and (if installed) the Green Hills Probe Administrator from the **Utilities** menu.

During development, you can use the MULTI Launcher as a convenient, centralized window manager. You can access any of your open MULTI windows from the **Windows** menu of the Launcher.

MULTI Workspaces

The MULTI Launcher allows you to create and use *workspaces*. A MULTI workspace is a virtual area where the tools, files, and actions required for a particular project can be organized, accessed, and executed.

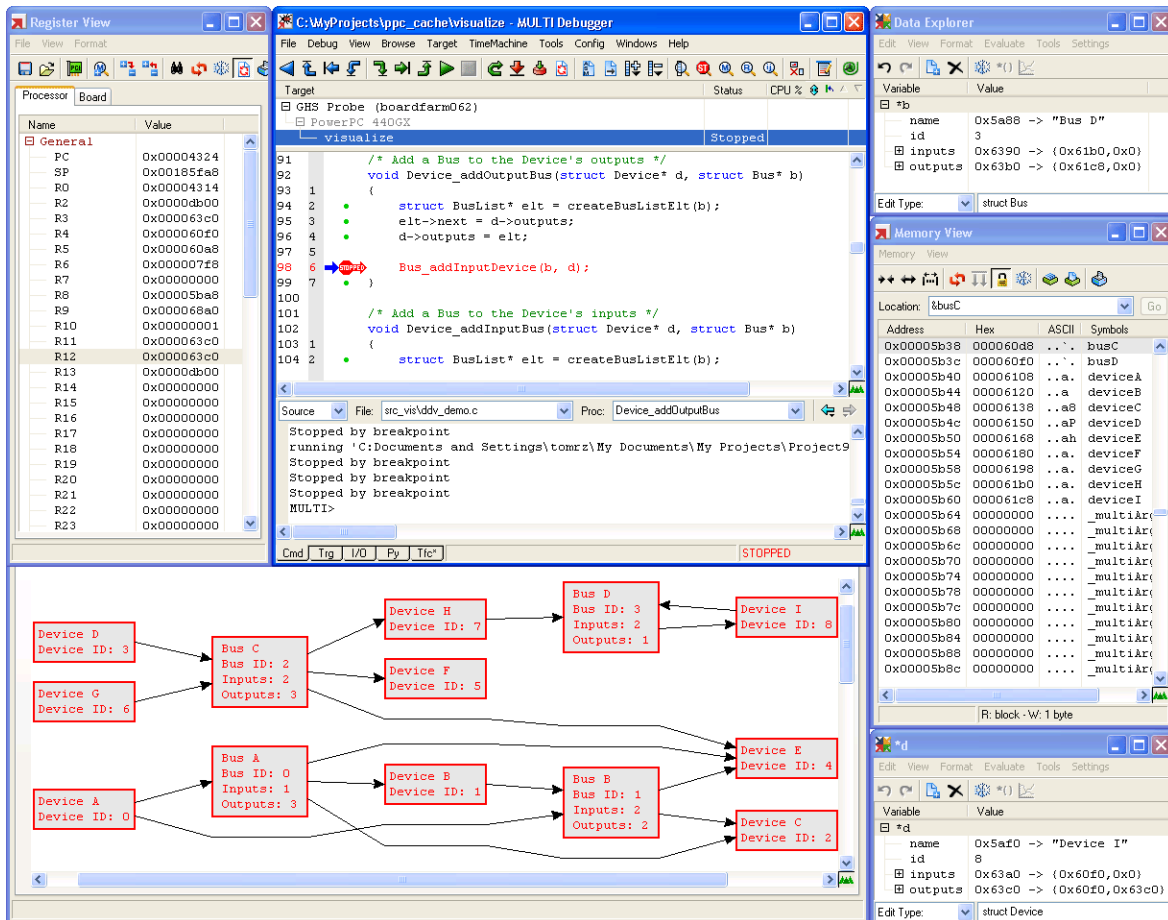


A workspace is typically created for each Top Project, and includes a working directory and a group of related *actions*—for example, opening a project in the MULTI Project Manager, connecting to a target, or performing a shell command. Actions are grouped into action sequences, so that a single mouse click can perform all the actions in the specified action sequence.

For more information, see Chapter 5, “Using MULTI Workspaces and Shortcuts” in the *MULTI: Editing Files and Configuring the IDE* book.

The MULTI Debugger

The MULTI Debugger is a powerful graphical debugger that supports source, assembly, and mixed-language debugging.



The MULTI Debugger's graphical interface makes it possible to view information about multiple processes and program elements simultaneously. The GUI also makes it easy to perform various debugging tasks, since most tasks can be invoked with a simple mouse click or keyboard shortcut. Most tasks can also be performed by typing commands into the Debugger's command pane.

The MULTI Debugger allows you to perform the following tasks quickly and easily:

- Download, execute, control, and debug embedded applications written in C, C++, assembly, or a combination of these languages.
- Browse, view, and search all aspects of your program code using graphical windows.
- View and edit variables, pointers, structures, registers, and memory ranges.
- Create, view, edit, and remove conditional breakpoints, non-intrusive tracepoints, data watchpoints, and Debugger Notes.
- View profiling, memory allocation, code coverage, and stack trace information.
- Analyze collected trace data.
- Interface seamlessly with the TimeMachine tool suite, the MULTI Editor, the MULTI Builder, the Eclipse environment, and with third-party editors and compilers.
- Perform multiprocess debugging through a single JTAG connection, even when those processes are running on multiple processors.
- Perform non-intrusive field debugging of live systems.

Accessing Online Help

Online help provides useful information about your MULTI tools and can be accessed in several different ways. The following sections describe how to access online help and how to use the Help Viewer.

Full Online Manuals

You can access an indexed, hypertext version of any MULTI manual by selecting it from the list that appears in the **Help** → **Manuals** menu. The MULTI Help Viewer opens on the manual specified.

You can also access manuals from the command line. Ensure that the MULTI installation directory is in your path and enter the following command:

```
helpview pathname | -m manual_name
```

where:

- *pathname* opens the **.chm** file specified in *pathname*. You must provide a full path.
- -m *manual_name* opens the manual *manual_name*. An example manual name is "MULTI: Debugging". If the manual name contains a space as in the preceding example, you must enclose it with quotation marks.

Context-Sensitive Help

Many MULTI windows and dialog boxes are linked to specific sections of the online manuals. To view the Help Viewer page that documents an active window or dialog box, press F1.

If you are using the MULTI Editor, you can also open context-sensitive help about a button or a menu item by selecting **Help** → **Identify** and then clicking the button or selecting the menu item.

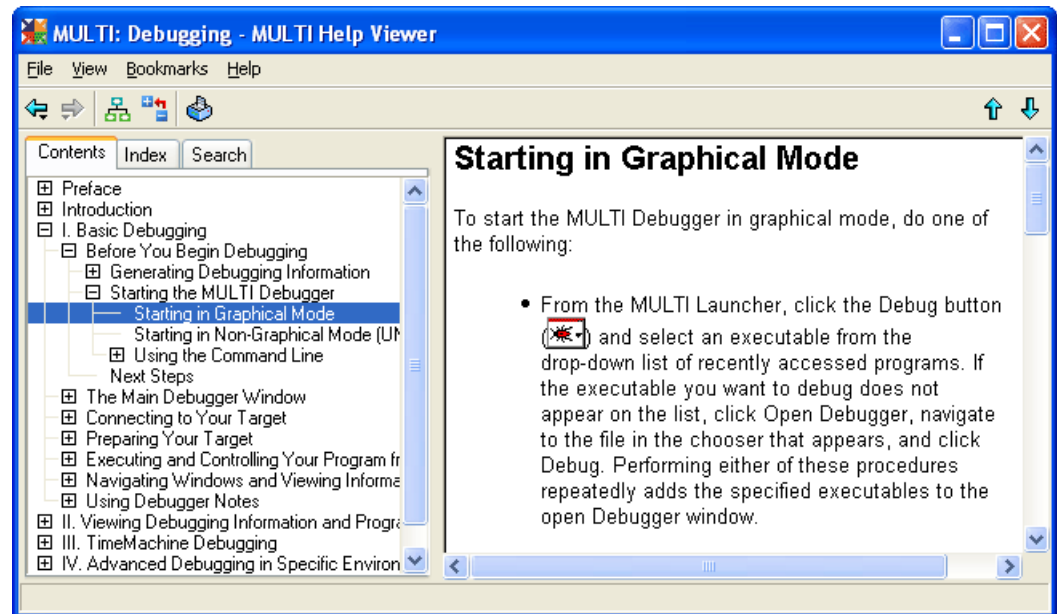
Command and Configuration Option Help

You can obtain help information about a specific MULTI Debugger command or MULTI configuration option by typing `help command_name` or `help configuration_option` in the Debugger command pane. This opens the Help

Viewer on documentation for the specified command or configuration option. You can also type `usage command_name` to print the basic syntax of a MULTI command to the command pane.

The Help Viewer Window

The following graphic displays the Help Viewer window.



The Help Viewer toolbar contains the following buttons, from left to right:

- **Back** (←) and **Forward** (→) — Returns to pages you have visited during the current help session. Click the button once for each page you want to revisit. Click and hold the button to select from a list of pages you have visited. These buttons function across books.
- **View All Subtrees** (📁) — Toggles highlighting of the current selection's sub-sections (if any) in the **Contents** tab, and toggles the display of help for the current selection's sub-sections (if any) in the view pane. To disable highlighting in the **Contents** tab and to display help for only the selected item, click this button again. When you open the Help Viewer, this button defaults to its last value.
- **Contract All Unselected** (📁) — Collapses hierarchies located in the **Contents** tab if the hierarchies do not contain any selections.

- **Print** () — Prints the help page displayed in the view pane. To create a help page consisting of mixed topics, hold down the Ctrl key while clicking separate topics. MULTI combines the separate topics and orders them by their location in the **Contents** tab.
- **Previous Page** () and **Next Page** () (located at far right) — Displays the previous or following section as ordered in the **Contents** tab. If the previous or next section includes sub-sections and the **View All Subtrees** button is enabled, the Help Viewer displays the sub-sections (if any) along with the section.

The MULTI Help Viewer contains two panes. The right-hand pane, or the *view pane*, displays the help page for your selection. The left-hand pane contains the following three tabs:

- **Contents** — Displays the parts, chapters, sections, and sub-sections for the manual you are viewing. Click a plus or minus icon to show or hide headings for embedded sections. Click a heading to display the associated help page in the view pane.
- **Index** — Displays index entries for the manual. To search the index entries, enter a word or words in the text box located at the top of the pane. MULTI returns results that contain the word(s) you typed. Click an index entry to display the corresponding help page in the view pane.
- **Search** — Provides an interface that allows you to search for specific words in the current manual or in all manuals.

Navigating Manuals in the Help Viewer

Navigating manuals in the MULTI Help Viewer is easy. This section describes different methods of finding information.

Viewing Previous and Next Sections

To display the previous or following section as ordered under the **Contents** tab, do one of the following:

- Click the  or  button located in the upper-right corner of the Help Viewer window.
- Select **View** → **Previous Contents Entry** or **View** → **Next Contents Entry**.

If the previous or next section includes sub-sections and the **View All Subtrees** button () is enabled, the Help Viewer displays the sub-sections (if any) along with the section.

Returning to Previously Viewed Help Pages

To return to pages you have visited during the current help session, do one of the following:

- Click the  or  button located in the left corner of the toolbar. Click the button once for each page you want to revisit. Click and hold the button to select from a complete list of pages you have visited during the current help session. These buttons function across books.
- Select **View** → **Back** or **View** → **Forward** from the menu bar. This feature functions across books.

Linking to Related Topics

The underlined links available in the view pane allow you to jump to pages related to the topic you are viewing.

If, when you click a link, a dialog box appears with the message:

You have requested information that the Help Viewer is unable to retrieve.

one of the following factors may be the cause:

- The manual that the link points to may not be included with your distribution. For a list of available manuals, look in the manuals directory (or directories) of your Green Hills Software distribution(s).
- If the link points to an INTEGRITY or u-velOSity manual, you may be using a version of the operating system that does not support links from MULTI help.
- If the link points to an INTEGRITY or u-velOSity manual, you may need to set the location of the installed OS distribution so that the Help Viewer can locate the relevant help files. For information about how to do this, see “Using MULTI with INTEGRITY or u-velOSity” in Chapter 2, “Installation” in the *MULTI: Getting Started* book.



Note If you are using MULTI with multiple target architectures (such as ARM and PowerPC), tools of the same version number should be installed into a single directory, regardless of differing architectures. Note, however, that links to target-specific books may yield unpredictable results in this situation.

Bookmarking Help Pages

You can use bookmarks to return to pages you have visited across MULTI sessions. To bookmark a page, select **Bookmarks** → **Bookmark This Page**.

You can bookmark a help page consisting of mixed topics by holding down the Ctrl key while clicking separate topics. MULTI combines the separate topics in the view pane and orders them by their location under the **Contents** tab. Bookmark the page as usual.

The **Bookmarks** menu lists all your bookmarks from across MULTI manuals and sessions. To return to a bookmarked page, do one of the following:

- Select **Bookmarks** and choose a bookmark.
- Select **Bookmarks** → **Manage Bookmarks**, click a bookmark, and click the **Show** button ().
- From any MULTI tool, select **Help** → **Bookmarks** and choose a bookmark.

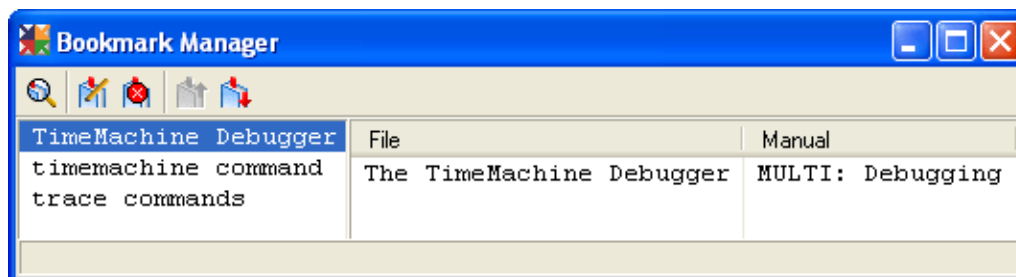
To rename, delete, or reorder existing bookmarks, launch the **Bookmark Manager** as described in the next section, “Managing Bookmarks” on page 12.

Managing Bookmarks

You can use the **Bookmark Manager** to rename and delete bookmarks, navigate to bookmarks, and modify the ordering of bookmarks in the **Bookmarks** menu.

To open the **Bookmark Manager**, do one of the following:

- In the Help Viewer, select **Bookmarks** → **Manage Bookmarks**.
- From any MULTI tool, select **Help** → **Bookmarks** → **Manage Bookmarks**.



The list on the left side of the **Bookmark Manager** displays the names of all the bookmarks that you have created. When a bookmark is selected, the file(s) and the manual marked by that bookmark are displayed in the right side of the window. The bookmark may reference multiple files (help pages) if, for example, you created the bookmark with multiple entries selected in the **Contents** tab or if you selected an index entry that pertained to multiple files.

The toolbar buttons in the **Bookmark Manager** allow you to perform the following operations on bookmarks:

-  — Display the selected bookmark(s) in the Help Viewer.
-  — Rename the selected bookmark.
-  — Delete the selected bookmark(s).
-  and  — Move the selected bookmark up or down in the list. The order of the bookmarks in the **Bookmarks** menu mirrors the order of the bookmarks in the **Bookmark Manager**.



Note The  and  buttons are available for use on multiple bookmarks. You can select multiple bookmarks from the left side of the window by dragging to select rows or by using Shift or Ctrl to select items.

Opening Additional Manuals from the Help Viewer

To open a new manual in the current window, do one of the following:

- Select **File** → **Open Manual** and choose from the list of installed manuals.
- Select **File** → **Open Manual File** and navigate to a particular file in the file chooser that appears.

To open a new manual in a new window, select **View** → **Open Other Manuals in New Window** and then open the manual as usual. When you open the Help

Viewer, the **Open Other Manuals in New Window** toggle option defaults to its last value.

Basic Debugging

Part I

Before You Begin Debugging

Chapter 2

This Chapter Contains:

- Generating Debugging Information
- Starting the MULTI Debugger
- Next Steps

This chapter describes the preliminary steps you must perform before debugging with the MULTI Debugger. These steps include generating debugging information when you compile your program, translating debugging information into a format the Debugger can use (if you used a third-party compiler), and starting the Debugger.



Note The instructions in this book assume that you have already installed MULTI. For step-by-step instructions on how to install MULTI and quickly create a new project, see the *MULTI: Getting Started* book.

Generating Debugging Information

The MULTI Debugger can either debug programs compiled with Green Hills compilers or programs compiled with third-party tools. The following sections briefly describe how to generate debugging information, including profiling and run-time error checking information, for your executables.

Programs Compiled with Green Hills Compilers

The MULTI Debugger uses special debugging information files generated by Green Hills compilers. To generate this information, you must set the MULTI Builder's **Debugging Level** appropriately (for example, to **MULTI**) or pass an appropriate compiler driver option (for example, **-G** or **-g**) when compiling your program. For more information about setting the debugging level, see the *MULTI: Building Applications* book for your target processor family.

When you build an executable with the **MULTI** debugging level or the **-G** or **-g** compiler driver option set, the Builder creates **.dnm** and **.dla** files. These output files contain symbolic debugging information used by the MULTI Debugger and have the same base name as your executable file. For example, for the executable **a.out**, the Builder generates the files **a.dnm** and **a.dla**. The **.dnm** file contains the debugging symbol table associated with your executable. The **.dla** file is an archive of **.dbo** files. The **.dbo** files contain debugging information about the object files and libraries associated with the executable. For more information about these files, see the *MULTI: Building Applications* book for your target processor family.

In most situations, the Debugger silently accesses the debugging information files created by Green Hills compilers. However, if the **.dnm** file the Debugger needs to access is older than the executable, a dialog box asks you if MULTI

should regenerate the **.dnm** and **.dla** files. Click the **Translate** button in the dialog box to generate new debugging information files. If the **.dnm** file does not exist, MULTI automatically generates new debugging information files.



Note If you move or copy an executable to another location, you also need to move or copy the associated **.dnm** and **.dla** files. When moving these files, either preserve the original time stamps of all the files or ensure that the **.dnm** file has a more recent time stamp than the executable's time stamp.



Note If you move an executable and its associated files, you may also need to re-map the *source root*. The source root is the path that MULTI prepends to each file's relative path. For more information, see the description of the **sourceroot** command in Chapter 7, "Configuration Command Reference" in the *MULTI: Debugging Command Reference* book.

Run-Time Error Checking Information

The MULTI Debugger provides wide-ranging run-time error checking information for programs compiled with Green Hills compilers. It does so by using a combination of compiler checks, special libraries, and Debugger commands.

To access run-time error checking information from the Debugger, you must set the appropriate MULTI Builder options or compiler driver options before compiling your program. Run-time error checking capabilities vary according to the type of processor and the operating system for which you are compiling. For information about which types of error checking are available for your target system and how to set your Builder or driver options, see the *MULTI: Building Applications* book for your specific target processor family.

Profiling Information

The MULTI IDE supports profiling for programs compiled with Green Hills compilers. To access profiling information from the MULTI Debugger, you must set the appropriate MULTI Builder or driver options before you compile your program. For detailed information about how to generate and view profiling information, see Chapter 15, "Collecting and Viewing Profiling Data".

Starting the MULTI Debugger

You can open the MULTI Debugger from within the MULTI IDE or from the command line. The sections below provide brief instructions about how to start the Debugger. For instructions about starting MULTI, see the *MULTI: Getting Started* book.

Starting in Graphical Mode

To start the MULTI Debugger in graphical mode, do one of the following:

- From the MULTI Launcher, click the **Debug** button () and select an executable from the drop-down list of recently accessed programs. If the executable you want to debug does not appear on the list, click **Open Debugger**, navigate to the file in the chooser that appears, and click **Debug**. Performing either of these procedures repeatedly adds the specified executables to the open Debugger window.
- From the MULTI Project Manager, select a compiled program and click the **Debug** button (). If the **Debug** button appears dimmed, you have not selected a compiled application. Performing this procedure repeatedly adds the specified executables to the open Debugger window.
- From the command line of your host system, start MULTI on a compiled program. For example, enter the command:

```
multi program
```

where *program* is an executable that has had some or all of its component modules compiled for debugging (as described in “Generating Debugging Information” on page 18). Entering this command repeatedly opens multiple Debugger windows.

For more detailed information about starting MULTI from the command line, including a description of common command line options to the Debugger, see “Using the Command Line” on page 22.

- From the command line of your host system, start MULTI on a special ELF file that was created from a Cafe OS core dump, in accordance with the Cafe SDK documentation. For example, enter the command:

```
multi core_file
```

where *core_file* is an special ELF file that was created from a Cafe OS core dump. Entering this command repeatedly opens multiple Debugger windows. For more detailed information about Cafe OS core dumps, see the Cafe SDK documentation.

- From the command line of your host system, start MULTI. Using a run-mode connection, attach MULTI to a target on which the target operating system is already running. For example, enter the command:

```
multi -connect="rtserv -port udp@myboard"
```

to connect to an operating system running on myboard. Entering such a command repeatedly opens multiple Debugger windows. For more information, see Chapter 18, “Run-Mode Debugging”.

Performing any of the above procedures once opens a MULTI Debugger window on the executable you specify. Performing a procedure more than once either opens a new Debugger window on the executable specified or adds the executable to an open Debugger window, as noted. For a description of the main features of the Debugger window, see Chapter 3, “The Main Debugger Window”.



Note On some UNIX systems, you can also start the Debugger in non-graphical mode. See the next section for more information.



Note If you receive a message indicating that debugging information is missing or out of date, click **Translate** in the error message dialog box to generate updated debugging information files. See “Generating Debugging Information” on page 18 for more information.

Starting in Non-Graphical Mode (UNIX only)

On certain versions of UNIX, you can run the MULTI Debugger in a non-graphical (or non-GUI) mode. Non-graphical mode does not support all of the features of the GUI mode and may not be sufficient for debugging some target systems.

To run the Debugger in non-GUI mode, start MULTI from the command line and use the **-nodisplay** option, as follows:

```
multi -nodisplay program
```

where *program* is an executable that has had some or all of its component modules compiled for debugging (as described in “Generating Debugging Information” on page 18).

Using the Command Line

You can start the Debugger from the command line by issuing the **multi** command and specifying the name of an executable program file for debugging. For example, enter the command:

```
multi [options...] [program | core_file]
```

where:

- *options* is any non-conflicting combination of command line options.



Note The most common options are listed in the following table. For a full list, see Appendix C.

- *program* is an executable that has had some or all of its component modules compiled for debugging, as described in “Generating Debugging Information” on page 18.
- *core_file* is a special ELF file that was created from a Cafe OS core dump, in accordance with the Cafe SDK documentation.

Common Command Line Options

-c *file*

-config *file* (Windows only)

-configure *file*

Configures MULTI using the information in the configuration file *file*. The **-config *file*** option is for Windows only. The other options apply to Windows and UNIX.

For more information, see “Creating and Editing Configuration Files” in Chapter 8, “Configuring and Customizing MULTI” in the *MULTI: Editing Files and Configuring the IDE* book.

-connect[=*target* | =*connection_method_name*]

Connects to the target debug server specified by *target*; connects using the specified Connection Method; or, if neither a target nor a Connection Method is specified, opens the **Connection Chooser**.

For more information about connecting to targets, see Chapter 4, “Connecting to Your Target”. For information about the **connect** command, which corresponds to this option, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.

Note: The Debugger ignores the deprecated `mode` argument in Connection Methods that specify it. To remove the `mode` argument from a MULTI 4 Connection Method, edit and save the Connection Method in MULTI 5. For more information, see the **connect** command in “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.

-data *offset*

Sets the default offset for all text addresses to *offset*, which is assumed to be in decimal unless preceded by `0x`. This option is only useful when debugging programs built with position-independent data; it should not be used in other kinds of programs.

-display *disp*

UNIX only

Specifies that the Debugger will open in the alternative display *disp*.

-h

Displays a concise description of all command line options.

-H

-help

Opens the MULTI Help Viewer on the *MULTI: Debugging* book.

-I *directory*

(This option is a capital letter “i.”)

Names an alternative directory for the Debugger to search for files in. You can specify multiple alternative directories by using multiple **-I *directory*** arguments. The Debugger searches alternative directories in the order given. If it does not find a file in the specified alternative directory or directories, it also searches the current directory.

-nodisplay

UNIX only

Specifies that the Debugger will open in non-GUI mode.

For more information, see “Starting in Non-Graphical Mode (UNIX only)” on page 21.

-norc

Causes MULTI to ignore **.rc** files on startup.

For more information, see “Using Script Files” in Chapter 8, “Configuring and Customizing MULTI” in the *MULTI: Editing Files and Configuring the IDE* book.

-p *file*

Runs the commands in the playback file *file* on startup.

For more information, see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

-P *pid*

Native targets only

Attaches to the process with the process identification number *pid*. The maximum number of processes you can specify is 256.

-r *file*

Records commands to the playback file *file*.

For more information, see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

-R *file*

Records commands and output to the playback file *file*.

For more information, see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

-RO file

Records output to the playback file *file*. (Commands are not recorded.)

For more information, see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

-text offset

Sets the default offset for all text addresses to *offset*, which is assumed to be in decimal unless preceded by `0x`. This option is only useful when debugging programs built with position-independent code; it should not be used with other kinds of programs.

Example 1. Starting the Debugger

The command:

```
multi a.out
```

opens the Debugger on the executable **a.out**.

Example 2. Starting the Debugger and Connecting to a Target

The command:

```
multi -connect=simppc a.out
```

opens the Debugger on the executable **a.out** and connects to the simulator **simppc**.

Using a Specification File

You can use a specification file to set up a default set of command line arguments, which are used when you open the Debugger on a specified executable. You can create default command line options for more than one executable in the same specification file.

Your specification file should have an **.mc** extension. To indicate the command line arguments associated with an executable, type the name of the executable followed by a space and the list of command line arguments, all on a single line. If you need to continue a list of arguments on a second line, the second line must begin with a tab. To create a list of commands for another executable, begin a new line and enter the executable name followed by a space and the list

of command line options. For example, a specification file that uses separate command line arguments when debugging **foo** and when debugging **bar** might contain the following lines.

Windows:

- `foo -I C:\usr\joebob -I C:\usr\foodir`
- `bar -text 10000 -data 40000`

UNIX:

- `foo -I /usr/joebob -I /usr/foodir`
- `bar -text 10000 -data 40000`

To use a specification file, pass the **-m file** option and launch the MULTI Debugger from the command line. This results in MULTI searching the specified file for a line that begins with the name of the executable on which you are opening the Debugger. If it finds a line that begins with the executable name, the Debugger uses the specified command line options on that line when it opens on the program. For example, if the specification file **albatross.mc** contains the lines given above, entering the command:

```
multi -m albatross.mc foo
```

has the same effect on your operating system as entering one of the following commands.

- Windows — `multi foo -I C:\usr\joebob -I C:\usr\foodir`
- UNIX — `multi foo -I /usr/joebob -I /usr/foodir`

Next Steps

The next chapter contains a detailed description of the main Debugger window and a summary of the key features that can be accessed from it.

While you can open a Debugger window and view your program code without connecting to your target, you will be unable to perform certain MULTI Debugger operations until you have connected MULTI to a target. These operations include viewing memory or registers and setting certain types of hardware breakpoints. For instructions about connecting MULTI to an embedded or simulated target, see Chapter 4, “Connecting to Your Target”. After you have established a connection, see Chapter 5, “Preparing Your Target” for information about running a program on your target.

The Main Debugger Window

Chapter 3

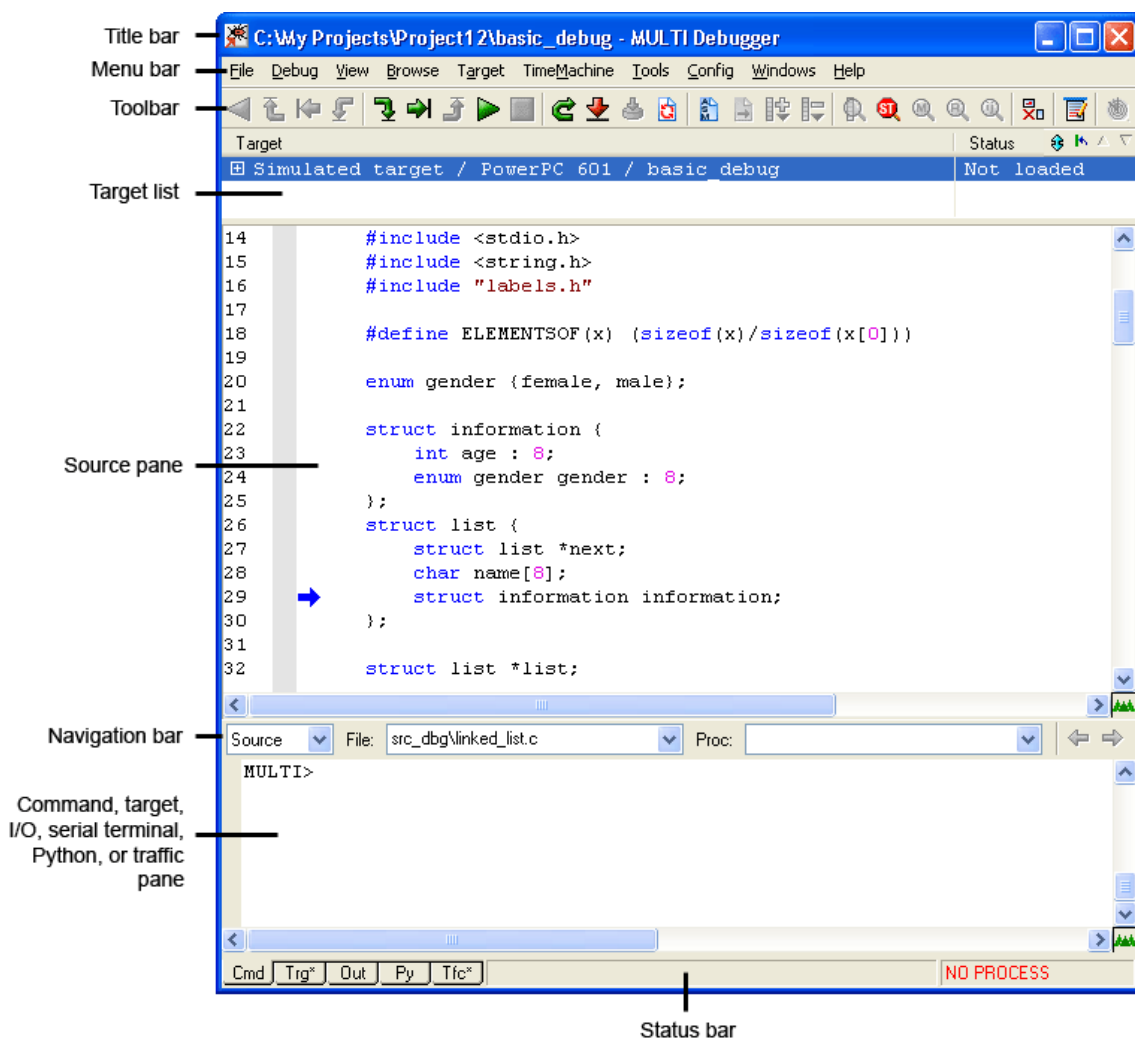
This Chapter Contains:

- Debugger Window Components
- Setting Up Your Debugging Environment
- The Target List
- The Source Pane
- The Navigation Bar
- The Cmd, Trg, I/O, Srl, Py, and Tfc Panes
- The Status Bar

This chapter describes the main Debugger window, which opens when you first start the MULTI Debugger. For instructions about starting the Debugger, see “Starting the MULTI Debugger” on page 20.

Debugger Window Components

The following graphic displays the main Debugger window. The topmost pane displays all items available for debugging and shows how these items are related. The middle pane displays source code for the item you are debugging. The bottom pane functions as a command prompt by default, but also allows you to view other panes.



The major components of the Debugger window are described below.

- *Title bar* — Displays the name of the executable being debugged. In this example, the executable name is **basic_debug**.
- *Menu bar* — Contains drop-down menus with the Debugger's most commonly used features. Specific menu options are documented throughout Parts I, II, and III of this manual, in the context of the tools or actions they are associated with. For a comprehensive description of Debugger menus and menu choices arranged by menu, see "Debugger Window Menus" on page 498. For information about how to customize menus, see "Configuring and Customizing Menus" in Chapter 8, "Configuring and Customizing MULTI" in the *MULTI: Editing Files and Configuring the IDE* book.
- *Toolbar* — Contains buttons with the Debugger's most commonly used features. Positioning your mouse pointer over a button displays the button's function. For a full description of each default button and its command equivalent, see "The Debugger Window Toolbar" on page 531. For instructions about how to add or remove buttons from the toolbar or customize buttons, see "Configuring the Debugger Toolbar" on page 538.
- *Target list* — Displays items that are currently available for debugging and shows the relationships among these items. For more information, see "The Target List" on page 33.
- *Source pane* — Displays the source code of your program. For more information, see "The Source Pane" on page 39. From the source pane, you can also open special windows to view variables, registers, target memory, the call stack, and other process and debugging information. For more information, see "Viewing Program and Target Information" on page 144.
- *Navigation bar* — Contains buttons and tools for changing the display mode of the source pane, browsing through the files and procedures of the program you are debugging, and jumping to previous or subsequent source pane views. For more information about these tools, see "The Navigation Bar" on page 43.
- *Command, Target, I/O, Serial Terminal, Python, and Traffic panes* — Functions as the command pane by default. The command pane accepts Debugger commands and displays the output of debugging activity. You can use the tabs beneath the pane to display a target pane, I/O pane, serial terminal pane, Python pane, or traffic pane instead of the command pane. For more information about each pane and how to switch among them, see "The Cmd, Trg, I/O, Srl, Py, and Tfc Panes" on page 44.

- *Status Bar* — Displays process status and various informational messages. For more information, see “The Status Bar” on page 53.

Setting Up Your Debugging Environment

When you are debugging, you can reuse the Debugger window and certain other windows or you can open new windows and debug different executables side by side. The following sections describe how to set up each of these debugging environments.

Reusing the Debugger Window

MULTI allows you to debug multiple items using one Debugger window. To debug a different executable in the current Debugger window, simply single-click the executable in the target list. The code for that executable appears in the source pane. Additionally, certain windows automatically update to display information relevant to the new executable. MULTI’s ability to remember this information as you switch between executables reduces the number of windows required for debugging, especially when you are debugging multi-core or multi-threaded systems.

MULTI can reuse the following windows:

- **Call Stack** window
- **Breakpoints** window
- **Memory View** window
- **Register View** window
- **Data Explorer**
- **Note Browser**

Opening Multiple Debugger Windows

The ability to open multiple Debugger windows is useful when you want to debug different executables side by side. To open a new Debugger window on an executable, perform one of the following actions in the target list:

- Double-click the executable.

- Right-click the executable and select **Open in New Window** from the shortcut menu that appears.



Note It is not possible to debug the same process in two windows.

If you open multiple Debugger windows, the background colors are different to help you distinguish among the windows. In the main Debugger window's target list, the names of executables that are open in another Debugger window are highlighted in the same color as the corresponding Debugger window.

The Target List

The target list displays all the items that are currently available for debugging. It may also list items that are not available for debugging (you cannot attach to any items that appear dimmed). The following sections provide information about repositioning the target list and about target list entries and terminology.



Note For the following sections, the term *executable* is used to mean any executable; thread; or INTEGRITY application, module, or kernel that you can select for debugging.

Repositioning and Hiding the Target List

By default, the target list appears in Debugger windows that have been launched from the command line, the MULTI Launcher, the Project Manager, etc., but not in Debugger windows that have been launched from other Debugger windows.

The first time you open the Debugger, the target list is located at the top of the window, above the source pane. If the target list takes up a lot of screen real estate (often the case when it contains many target list entries), you may want to move it to the left side of the Debugger window. To do so, click the **Move target list to left** button () , which is located to the right of the **Status** column. To move it back to the top of the window, click the **Move target list to top** button (). MULTI remembers the location of the target list across sessions.

When you change the target list location, MULTI attempts to preserve the width of the source pane by resizing the window. If the source pane area is too small, MULTI does not change the window size.



Tip As an alternative to moving the target list, you can maximize the target list area and then debug every executable in a separate window. To maximize the target list area, click the **Move splitter down** button () or the **Move splitter right** button () (depending on the position of the target list). For information about debugging executables in separate windows, see “Opening Multiple Debugger Windows” on page 32.

By default, the target list is auto-sized when it occupies the top pane of the Debugger window, but you can also size it manually. To do so, drag the splitter that separates the target list from the source pane. To revert to the default auto-sizing behavior, click the **Auto-size splitter** button () , which is located to the right of the **Status** column.

To hide the target list completely, click the **Move splitter up** button () or the **Move splitter left** button () (depending on the position of the target list), or drag the splitter.

Debugging Target List Items

The content of the source pane varies depending on the item you select in the target list. The following list describes what appears in the source pane when you select a particular item in the target list.

- An executable — The executable’s source code appears in the source pane.
- An OSA AddressSpace — The source pane is empty. OSA AddressSpaces are located under CPU nodes, and their names begin with **OSA**.
- An OSA master process — The kernel’s source code appears in the source pane. The master process is listed under the CPU node of a freeze-mode connection. For more information, see “Master Debugger Mode” on page 458.
- An OSA task — The task’s source code appears in the source pane. OSA tasks are located under OSA AddressSpaces (AddressSpaces whose names begin with **OSA**). MULTI refreshes OSA tasks in the target list when the **OSA Explorer** is refreshed or when trace data is retrieved. If you have frozen or closed the **OSA Explorer** and disabled the retrieval of trace data, MULTI does not refresh OSA tasks when the target halts. For more information, see “Debugging in Freeze Mode” on page 457 and “Task Debugger Mode” on page 459.

- A run-mode AddressSpace — If you are using INTEGRITY version 10 or later, the AddressSpace’s source code appears in the source pane. Otherwise, the source pane is empty. Run-mode AddressSpaces are located under a CPU node of a run-mode connection to the target, or they are located under an application (kernel image or dynamic downloaded module), which is in turn located under a CPU node. A run-mode connection to a target is usually denoted by **Run mode target**. For more information, see “Run-Mode AddressSpaces” on page 425.
- A run-mode task — The task’s source code appears in the source pane. Run-mode tasks are located under run-mode AddressSpaces. For information about run-mode debugging, see Chapter 18, “Run-Mode Debugging”.

Any commands you give via menu selections, buttons, or the command pane are performed on the item that is selected in the target list (if applicable).

For information about how the above items are arranged in the target list, see the next section.

The Target List Display

Related executables, CPUs, and debug servers are arranged hierarchically on several lines or compressed onto one line. Both views illustrate the relationships among items. Which items appear and the style of grouping is dependent upon your current environment. The two views available are described below:

- Compressed — The Debugger compresses all related items onto a single line when the resulting line contains only one of each item. For example, suppose you are connected to a PowerPC target that is running a kernel. The debug server, CPU, and kernel are all compressed onto one line, as shown next.

Target	Status	CPU %	
Simulated target / PowerPC 860 (PowerQUICC) / kernel	Running		

- Hierarchical — The Debugger arranges related items as a hierarchy when the resulting hierarchy contains multiple occurrences of any item. For example, if a kernel contains multiple tasks, the items are arranged as shown next.

Target	Status	CPU %
Run mode target		
PowerPC 860 (PowerQUICC)		
kernel		100.00
Initial	Zombied	0.00
Idle	Running	100.00
ResourceManager	Pended	0.00
LoaderTask	Pended	0.00
MULTILoadAgent	Host IO	0.00

In both compressed and hierarchical view mode, you can change the display by expanding or collapsing nodes. If you collapse a node that encompasses multiple items of the same type (for example, an address space that encompasses multiple tasks), the source pane is empty and the target list does not display status or CPU information for the parent object. Displaying a single status for the parent object would be ambiguous in this case because multiple child objects, each of which have a valid status, are encompassed by the parent object.

As the previous samples illustrate, executables appear after the CPU that they are associated with. The CPU in turn appears after the appropriate debug server.

Target Terminology

The following list defines the phrases that are used in the **Target** section of the target list.

- **Direct hardware access** — Appears under a CPU name in the target list when you are not actively debugging an executable on that CPU. For information about preparing to debug an executable on your target, see Chapter 5, “Preparing Your Target”.
- **Unconnected Executables** — The executables that appear under this heading are not associated with a connection. For more information, see “Associating Your Executable with a Connection” on page 87.

The Status Column

The Debugger lists the statuses of certain items in the target list's **Status** column and automatically updates these statuses when appropriate. The following table defines common statuses. Not all statuses are supported by all operating systems.

Status	Meaning
DebugBrk	The task has been stopped due to a MULTI breakpoint or single-stepping.
Execing	The process has just performed an <code>exec()</code> call—a kernel call via a software interrupt.
Exited	The task has exited (usually by calling <code>exit()</code>).
GrpHalt	The task, which is part of a task group, has been halted by a group breakpoint. For more information see, “Working with Task Groups in the Task Manager” on page 428.
Halted	The run-mode task has been halted by the operating system.
HostIO	The task is writing to or reading from a file on the host system or the I/O pane. This status may also signify that the target process is waiting for input from <code>stdin</code> . In this case, select the process in the target list, click in the I/O pane, and type the input you want the process to receive.
No TimeMachine Data	Trace data was not saved for the loaded task.
Not loaded	The executable has not been downloaded or flashed onto a target or is not currently running on the target system. For information about loading the executable on a target, see “Preparing Your Target” on page 90.
Pended	The exact meaning is OS-specific. In general, this status signifies that the run-mode task is waiting for something to happen, such as the release of a semaphore or message on a socket.
Running	The task is running.
Stopped	The task is stopped.
SysHalt	All tasks on the target system are halted, and the target is frozen.
Zombied	The task has exited (usually by calling <code>exit()</code>), but data structures describing the task still exist on the target.

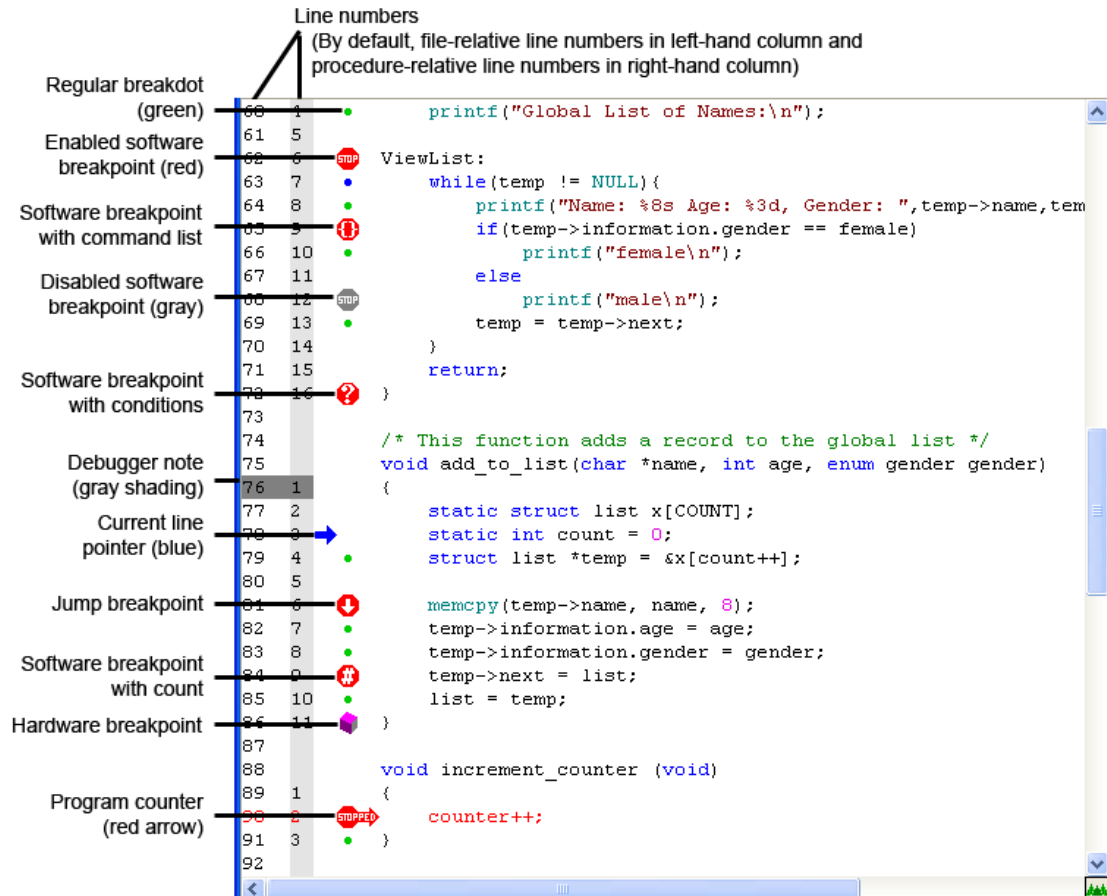
The CPU % Column

In specific debugging environments, a **CPU %** column appears in the Debugger window's target list. It displays the percentage of the CPU that each task and AddressSpace is currently using. For more information, see "Profiling All Tasks in Your System" on page 447.

Target	Status	CPU %	
[-] Run mode target			▲
[-] PowerPC 860 (PowerQUICC)			
[-] pizza-kernel		0.24	
Initial	Zombied	0.00	
Idle	Running	0.24	
[-] mphonecompany / Initial	Created	0.00	
[-] mengineer / Initial	Created	0.00	
[-] minformation / Initial	Created	0.00	
[-] mpizzahut / Initial	Created	0.00	
System		99.76	▼

The Source Pane

The source pane displays the source code of your program. Your program's source code can be in C, C++, or assembly language.



The main features of the source pane are described below.

- *Source pane display modes* — The source pane can display your program code in high-level language, mixed high-level and assembly language, or assembly language. You can use the Display Mode Selector in the navigation bar or the **View** → **Display Mode** menu option to switch among these display modes. For more information, see “Source Pane Display Modes” on page 41.
- *Line numbers* — By default, file-relative line numbers appear in the left-most column of the source pane, and procedure-relative line numbers appear to the right of the file-relative numbers. You can configure the Debugger to

display either, both, or neither of the columns or to display the columns in opposite orientation. See “Source Pane Line Numbers” on page 42.

- *Breakdots* — A breakdot (•) is a small dot located to the left of any line of code that corresponds to an executable instruction. You can set a software breakpoint simply by clicking a breakdot. You can also use breakdots to set hardware breakpoints or tracepoints, if your target supports them. For more information, see “Breakdots, Breakpoint Markers, and Tracepoint Markers” on page 103.
- *Breakpoint markers* — A breakpoint marker is an icon that appears in place of a breakdot when you set a breakpoint. Hardware breakpoints are indicated by a small purple cube (🟪). Software breakpoints are indicated by a small red stop sign (🛑). The stop sign may contain a symbol indicating that certain conditions are associated with the breakpoint. If a breakpoint is disabled, the breakpoint marker is gray. For more details, see “Breakdots, Breakpoint Markers, and Tracepoint Markers” on page 103.
- *Debugger Notes* — Debugger Notes are free form notes that can be associated with any line of code. The line number column(s) of lines associated with Debugger Notes are shaded gray. For more information, see Chapter 8, “Using Debugger Notes”.
- *Current line pointer* — The blue current line pointer (➡) jumps to any line you type in the command pane, select with a mouse click, or navigate to with commands. If you run or step a process, the pointer jumps to the location where the process halts. Many Debugger commands operate at the line pointer location if no other location is specified.

When you open a program, the line pointer is located at the entry point. If you stop a running process, the line pointer jumps to the code where the process has stopped. You can also navigate to other locations easily. For more information about how to navigate to different locations, see Chapter 7, “Navigating Windows and Viewing Information”.

- *Program counter (PC) pointer* — The program counter (PC) pointer is a red arrow marked with the word **STOPPED** (🛑). When a process stops, the PC pointer identifies the line that will execute next when you resume the process. When the Debugger stops on a conditionally not-executed instruction, or when the PC is not in the selected image, the PC pointer turns gray (🛑). When you step backward in the Debugger or launch TimeMachine, the PC pointer turns blue (➡). When you move up or down the call stack, the PC pointer becomes a hollow arrow outlined in red (⇨).

- *Shortcut menus* — Shortcut menus are available when you right-click a source line, a breakpoint, a procedure, a variable, a type, or a member of a type. See “Source Pane Shortcut Menus” on page 543 for more specific information about these menus.



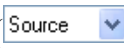
Note You can open other windows to view details about certain program elements, such as variables, by double-clicking them in the source pane. For more information, see “Viewing Program and Target Information” on page 144.

Source Pane Display Modes

The Debugger source pane has three viewing modes for code display:

- **Source only** — Displays only high-level source code. This is the default display mode.
- **Interlaced assembly (Mixed)** — Displays high-level source code interlaced with the corresponding assembly instructions.
- **Assembly only** — Displays only assembly code.

You can change the source pane display mode in any of the following ways:

- From the Display Mode Selector’s drop-down menu, select **Source**, **Mixed**, or **Assem**. The Display Mode Selector () appears on the navigation bar.
- Press the **Assembly** button () located on the toolbar to toggle between source only and interlaced assembly view.
- From the **View** → **Display Mode** menu, select **Source Only**, **Interlaced Assembly**, or **Assembly Only**.
- Use the **assem** Debugger command. For information about this command, see Chapter 9, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book.

When assembly code is displayed (interlaced assembly or assembly only mode), the hexadecimal address of each assembly instruction appears in the source pane. When the process stops at a line in the high-level source code, the PC pointer () appears at the first assembly instruction associated with that high-level source line. Note that not all high-level source lines generate executable code.

If no high-level source code is available to the Debugger, or if the module was not compiled with sufficient debugging information (for example, if it was compiled without the **-G** or **-g** driver option or **MULTI** debugging level set), the Debugger only displays assembly code, regardless of the display mode setting.

Source Pane Line Numbers

The Debugger supports both procedure-relative and file-relative line numbers. You can configure the Debugger source pane to display either, both, or neither of these line numbers. To configure the appearance of procedure-relative and file-relative line numbers, select **Config** → **Options** → **Debugger** tab. From the **Line numbers in source pane** text box, choose **No Number**, **File Number**, **Proc Number**, or **Both Numbers**.

When both file-relative and procedure-relative line numbers are displayed, they appear as two separate columns on the left side of the source pane. By default, file-relative line numbers appear in the left-most column, and procedure-relative line numbers appear to the right of the file-relative numbers and to the left of the breakdots. You can change the orientation of the column display with one of the following methods:

- Select **Config** → **Options** → **Debugger** tab. If you check the **Use procedure relative line numbers (vs. file relative)** box, file-relative numbers appear in the left column and procedure-relative numbers in the right column. This is the default setting. If you clear this check box, file-relative numbers appear in the right column and procedure-relative numbers in the left column. This option also affects how line numbers in Debugger commands are interpreted. See “Specifying Line Numbers” in Chapter 2, “Using Debugger Commands” in the *MULTI: Debugging Command Reference* book.
- Use the **configure** command to set the option **ProcRelativeLines** to **on** or **off** (for more information, see “Using the configure Command” in Chapter 8, “Configuring and Customizing MULTI” in the *MULTI: Editing Files and Configuring the IDE* book). For example:

```
> configure ProcRelativeLines on
```

Setting this option to **on** causes file-relative numbers to appear in the left column and procedure-relative numbers to appear in the right column. This is the default setting. Setting this option to **off** causes file-relative numbers to appear in the right column and procedure-relative numbers to appear in

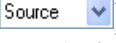
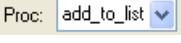
the left column. This option also affects how line numbers in Debugger commands are interpreted. See “Specifying Line Numbers” in Chapter 2, “Using Debugger Commands” in the *MULTI: Debugging Command Reference* book.

The Navigation Bar

With the navigation bar, you can change the display mode of the source pane; browse through the output, files, and procedures of the program you are debugging; and jump to previous or subsequent source pane views. The navigation bar is located below the source pane and above the command pane in the Debugger window.



The navigation bar contains the following buttons and tools.

- Display Mode Selector () — Changes the source pane’s display mode to source only (**Source**), interlaced assembly (**Mixed**), or assembly only (**Assem**). For more information, see “Source Pane Display Modes” on page 41.
- File Locator () — Shows the name of the file currently displayed in the Debugger and allows you to quickly navigate to other files in your program. For a full description of how to use this tool, see “Using the File Locator” on page 139.
- Procedure Locator () — Shows the name of the procedure currently displayed in the Debugger and allows you to quickly navigate to other procedures in your program. For a full description of how to use this tool, see “Using the Procedure Locator” on page 140.
- **Back** () and **Forward** () buttons — Allow you to jump to files and procedures you have viewed before or after the current view. For more information, see “Using Navigation History Buttons” on page 142.

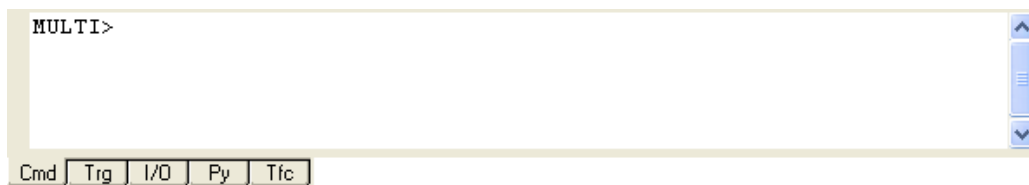
You can change the relative sizes of the source pane and the command pane by dragging the navigation bar up or down.



Note The pane-switching buttons that appeared on the navigation bar in previous releases and that allowed you to switch between the command, target, and I/O panes have been replaced by tabs under the command pane. For more detailed information, see the next section.

The Cmd, Trg, I/O, Srl, Py, and Tfc Panes

The pane located below the navigation bar in the Debugger window functions as the command pane by default. But it can also display a target pane, I/O pane, serial terminal pane, Python pane, or traffic pane. The availability of individual panes depends upon the current environment.



To change which pane is displayed in this area, click the appropriate tab located in the lower-left corner of the pane. The following table describes each tab and pane.

Tab	Pane	Description
Cmd	Command	Allows you to enter Debugger commands and view the output of debugging activity. This is the default pane. For more information about using the command pane, see “The Command Pane” on page 45.
Trg	Target	Allows you to send commands to your debug server. This tab and pane are only available if your debug server supports this feature and you are connected to a target. For more information about using the target pane, see “The Target Pane” on page 48.
I/O	I/O (input/output)	Provides basic I/O for the process you are debugging. This tab and pane are only available if your debug server supports this feature and you are connected to a target. If input is not possible for the currently selected target, the pane is labeled Out . For more information about using the I/O pane, see “The I/O Pane” on page 48.

Tab	Pane	Description
Srl	Serial terminal	Allows you to send commands to a serial port and receive input back from it. This tab and pane are only available if you are connected to an active serial port. For more information about using the serial terminal pane, see “The Serial Terminal Pane” on page 49.
Py	Python	Allows you to execute Python statements and view MULTI-Python output. This pane only remembers Python history for the current Debugger’s debugging session. For more information about using the Python pane, see “The Python Pane” on page 50.
Tfc	Traffic	Provides messages detailing interactions between MULTI and your target. This pane only contains meaningful messages if you have connected to a target. For more information about using the traffic pane, see “The Traffic Pane” on page 50.

If a pane is not available in your current context, the tab for that pane does not appear. If a tab is not selected but new output is available in the corresponding pane, the tab is marked with an asterisk (*). The asterisk indicates that output is available for viewing.



Tip For information about limiting the number of scroll back lines available in these panes, see the **C textSize** option in “Other Debugger Configuration Options” in Chapter 9, “Configuration Options” in the *MULTI: Editing Files and Configuring the IDE* book.

The Command Pane

The command pane accepts Debugger commands for the process you are debugging and displays the output of those commands. By default, the command pane also displays color-coded output from the target and I/O panes. When the **Cmd** tab is selected, the command pane appears below the navigation bar in the Debugger window. This is the default tab setting.



If the **Cmd** tab is not selected, but new output is available in the command pane, the tab is marked with an asterisk (*).

In the Debugger window, most keystrokes you make go into the command pane, unless you have clicked in the File Locator or the Procedure Locator text box. The command pane automatically evaluates expressions using the syntax of the source code language currently being debugged. For general information about using Debugger commands and for a detailed description of the Debugger commands that can be entered in the command pane, see the *MULTI: Debugging Command Reference* book.



Note The default prompt in the command pane is **MULTI>**. You can specify a different prompt by entering the command **configure prompt** *new_prompt* in the command pane, or by setting the **Command pane prompt** configuration option. For information about the **configure** command, see “General Configuration Commands” in Chapter 7, “Configuration Command Reference” in the *MULTI: Debugging Command Reference* book. For information about the **Command pane prompt** option, see “The Debugger Options Tab” in Chapter 9, “Configuration Options” in the *MULTI: Editing Files and Configuring the IDE* book. For information about saving your prompt, see “Saving Configuration Settings” in Chapter 8, “Configuring and Customizing MULTI” in the *MULTI: Editing Files and Configuring the IDE* book.

The Debugger keeps a history of all the Debugger commands entered in the command pane. You can use history commands, such as the **!** and **!!** commands, to search the command history and execute previously entered commands. For more information, see “History Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book. You can also record and play back sequences of Debugger commands. For more information, see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

Command Pane Shortcuts

In the Debugger window, some key combinations, such as Ctrl + UpArrow, function as shortcuts that invoke actions in the command pane. The following table lists common shortcuts that affect the command pane. For a comprehensive list of shortcuts, see “Command Pane Shortcuts” on page 561.

Shortcut	Effect
UpArrow	Retrieve the previous command from the command history, moving back in the list. Press repeatedly to retrieve older commands.
DownArrow	Retrieve the next command from the command history, moving forward in the list. Press repeatedly to retrieve more recent commands.
Ctrl + UpArrow	Scroll up four lines.
Ctrl + DownArrow	Scroll down four lines.
Ctrl + U	Cuts to the beginning of the current line or the selection.
Tab	Accept auto-completion of the current word.
Ctrl + P	Attempt to auto-complete the current string, working backwards through the command history for commands beginning with the typed characters.
Ctrl + D	Display a list of possible completions for the current word.
F6	Cycle between the available panes in the command pane area. For more information, see “The Cmd, Trg, I/O, Srl, Py, and Tfc Panes” on page 44.
Select text with the left mouse button	Copy a string (UNIX only).
Click with the right mouse button	Open a shortcut menu. For a full description of the shortcut menu, see the “The Command Pane Shortcut Menu” on page 552.



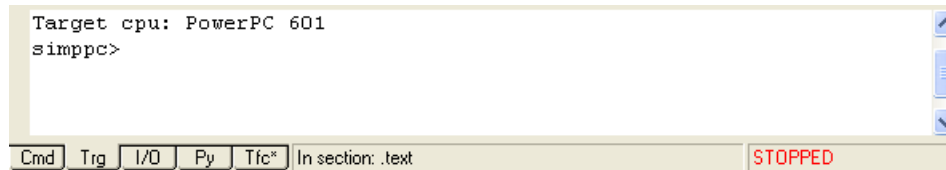
Note The procedures for copying, cutting, and pasting text in the command pane differ slightly from the procedures used in other windows. For more information, see “Selecting, Copying, and Pasting Text in the Main Debugger Window” on page 135.

For a full list of default shortcuts, see Appendix B. For information about reconfiguring the MULTI IDE’s default shortcut keys and mouse clicks, see “Customizing Keys and Mouse Behavior” in Chapter 8, “Configuring and

Customizing MULTI” in the *MULTI: Editing Files and Configuring the IDE* book.

The Target Pane

The target pane allows you to send commands to your debug server. When the **Trg** tab is selected, the target pane appears below the navigation bar in the Debugger window.

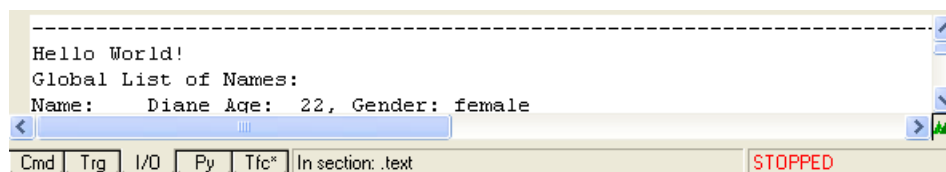


If the **Trg** tab is not selected, but new output is available in the target pane, the tab is marked with an asterisk (*).

This tab and pane are only available if your debug server supports this feature and you are connected to a target. For more information about debug servers, see the *MULTI: Configuring Connections* book for your target processor family. For instructions about connecting to your target, see Chapter 4, “Connecting to Your Target”.

The I/O Pane

The I/O pane provides basic I/O for the process you are debugging. When the **I/O** tab is selected, the I/O pane appears below the navigation bar in the Debugger window.



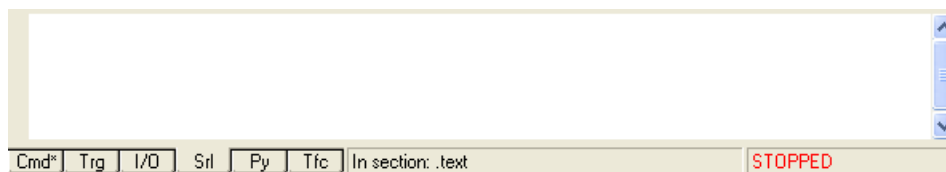
If the **I/O** tab is not selected, but new output is available in the I/O pane, the tab is marked with an asterisk (*). If input is not possible for the currently selected target, the pane is labeled **Out**.

This tab and pane are only available if your debug server supports this feature and you are connected to a target. For more information about debug servers,

see the *MULTI: Configuring Connections* book for your target processor family. For instructions about connecting to your target, see Chapter 4, “Connecting to Your Target”.

The Serial Terminal Pane

The serial terminal pane allows you to send commands to a serial port and receive input back from it. When the **Srl** tab is selected, the serial terminal pane appears below the navigation bar in the Debugger window.



If the **Srl** tab is not selected, but new output is available in the serial terminal pane, the tab is marked with an asterisk (*).

This tab and pane are only available if you are connected to an active serial port. There are two ways to connect to a serial port:

- Select **Tools** → **Serial Terminal**. From the submenu that appears, you can open recent connections or create a new serial connection using the **Serial Connection Chooser**. For more information, see “Using the Serial Connection Chooser” on page 484.
- In the Debugger command pane, issue the **serialconnect** command with appropriate arguments. For information about this command, see “Serial Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.



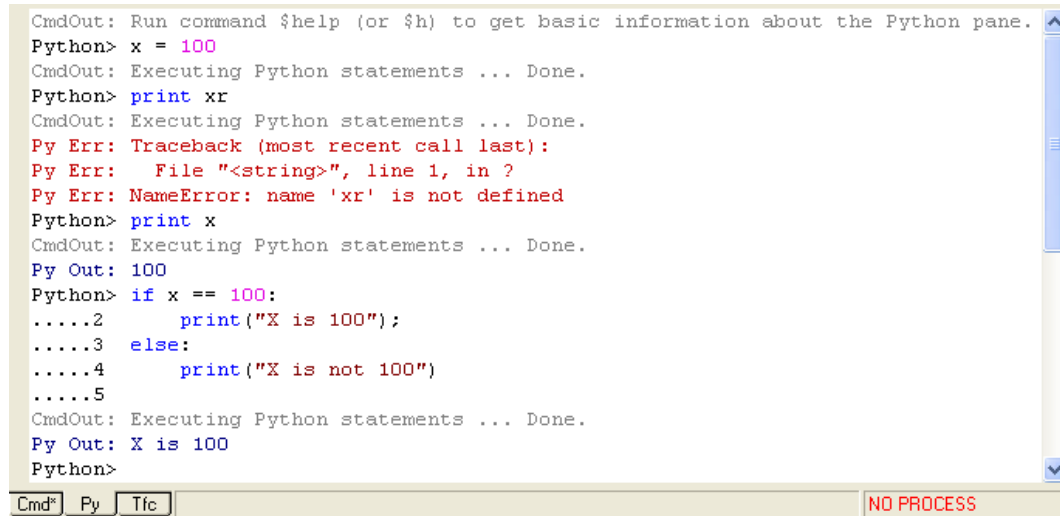
Note Because there is only one serial terminal pane per Debugger window, only one serial connection can be maintained and viewed per Debugger window. If you require multiple serial connections at the same time, see Chapter 20, “Establishing Serial Connections”, for alternate ways to use serial connections.

Once you have established a serial connection, the serial terminal pane is available. Input to this pane is sent to the serial port; output from the serial port is sent to this pane.

The serial terminal pane provides full **VT100** support.

The Python Pane

The Python pane allows you to execute Python statements and view MULTI-Python output. When the **Py** tab is selected, the Python pane appears below the navigation bar in the Debugger window.



```
CmdOut: Run command $help (or $h) to get basic information about the Python pane.
Python> x = 100
CmdOut: Executing Python statements ... Done.
Python> print xr
CmdOut: Executing Python statements ... Done.
Py Err: Traceback (most recent call last):
Py Err:   File "<string>", line 1, in ?
Py Err: NameError: name 'xr' is not defined
Python> print x
CmdOut: Executing Python statements ... Done.
Py Out: 100
Python> if x == 100:
.....2     print("X is 100");
.....3 else:
.....4     print("X is not 100")
.....5
CmdOut: Executing Python statements ... Done.
Py Out: X is 100
Python>
```

At the bottom of the pane, there are tabs for **Cmd***, **Py**, and **Tfc**. The **Py** tab is selected. To the right of the tabs, it says **NO PROCESS**.

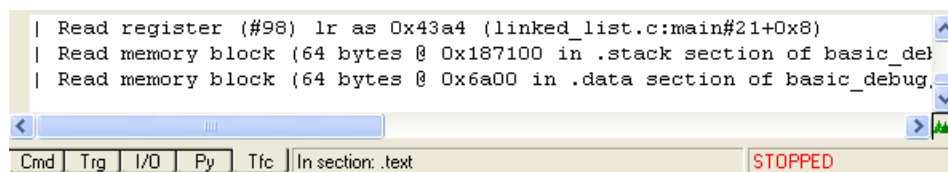
If the **Py** tab is not selected, but new output is available in the Python pane, the tab is marked with an asterisk (*).

This tab and pane are always available, but the Python pane only remembers Python history for the current Debugger’s debugging session. For more information about this pane, see “MULTI-Python Interfaces” in Chapter 3, “Introduction to the MULTI-Python Integration” in the *MULTI: Scripting* book.

To open the Python pane as a separate window, right-click in the Python pane area and select **Show Separate Py Window** from the shortcut menu.

The Traffic Pane

The traffic pane displays messages that detail the interactions between MULTI and your target. When the **Tfc** tab is selected, the traffic pane appears below the navigation bar in the Debugger window.



```
| Read register (#98) lr as 0x43a4 (linked_list.c:main#21+0x8)
| Read memory block (64 bytes @ 0x187100 in .stack section of basic_de
| Read memory block (64 bytes @ 0x6a00 in .data section of basic_debug.
```

At the bottom of the pane, there are tabs for **Cmd**, **Trg**, **I/O**, **Py**, and **Tfc**. The **Tfc** tab is selected. To the right of the tabs, it says **STOPPED**.

If the **Tfc** tab is not selected, but new output is available in the traffic pane, the tab is marked with an asterisk (*).

This tab and pane are always available. However, you will only see meaningful messages if you have connected to a target during the current MULTI session.

Viewing traffic information may be useful if you see unexpected results while working in MULTI or if your target crashes. You can examine traffic pane messages to see what commands MULTI has issued to the target and to see how the target has responded.

Each line of information in the traffic pane is referred to as a *traffic report*. When you are debugging native UNIX and Windows processes, each traffic report begins with `pid`—for “process ID”—followed by the process ID number. When you are debugging INTEGRITY tasks, each traffic report begins with `task` followed by a hexadecimal number. These numbers and either `pid` or `task` are always enclosed in parentheses. The target gives every process a new process ID number and every task a new task number, making it easy to tell processes and tasks apart. If the target does not know the process ID or task number, or if none exists, the traffic report does not list any number.

The traffic pane displays MULTI commands that result in target traffic, requests that MULTI makes to the target, and resulting target responses or actions. The following example typifies what you might see in the traffic pane.

Suppose you are debugging an ARM stand-alone program on a SimARM simulator. The simulator just executed the instruction `MOV R0, 20`, and you type the following into MULTI’s command pane.

```
MULTI> print $r0
```

The following result appears.

```
0x00000014
```

If you view the traffic pane contents, you see the following.

```
MULTI command: print $r0
| Read register (#0) r0 as 0x14
```

As a result of you typing the **print** command, MULTI asked the target (in this case, SimARM) to read register R0.

Most MULTI buttons and menu items correspond to MULTI commands. Therefore, if you click a button or select a menu item, the traffic pane displays the command associated with your action. For example, suppose you click the  button or press F10 to single-step a Linux x86 native process. If you view the traffic pane contents, you might see something like the following.

```
MULTI command: __ntwcommand next
\_ MULTI command: n
| (pid 22434) Set software breakpoint at 0x8049448 (foo.cc:main#12)
| (pid 22434) Resume process at 0x8049437 (foo.cc:main#11), in source step
| (pid 22434) Read memory block (64 bytes @ 0xbfece180)
| (pid 22434) Remove breakpoint from 0x8049448 (foo.cc:main#12)
```

The  button and the F10 key are linked to MULTI's **n** command. As a result, MULTI executes the **n** command when you click  or press F10. The **n** command causes MULTI to set a temporary breakpoint at the next source line (in this case, line 12 of `main()`), and then resume the process from the current source line (in this case, line 11 of `main()`). When it reaches line 12, it removes the temporary breakpoint.

In addition to displaying interactions between your target and MULTI, the traffic pane also displays the chain of commands (if any) that causes target traffic. You might not explicitly specify multiple commands; however, some MULTI commands execute others as a part of their operation. For example, suppose that while a process is running, you enter the **tog** command to inactivate a breakpoint. You might see something like the following.

```
MULTI command: tog off main#2
| (pid 22807) Halt remote process (stop thread)
| (pid 22807) Status is StatHalted
| (pid 22807) Remove breakpoint from 0x80491bf (foo.cc:main#2)
\_ MULTI command: c
| (pid 22807) Resume process at 0xb7f85402
| (pid 22807) Status is StatRunning
```

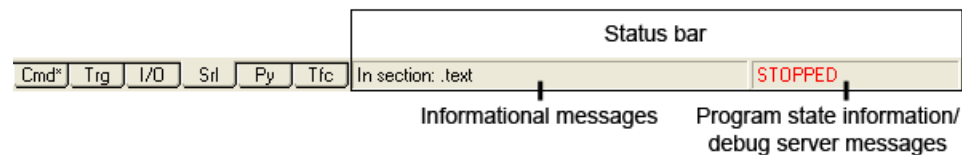
In this case, MULTI halts the process and later executes the **c** command to continue it. When one command executes another as shown, the commands appear nested in the traffic pane. A backslash followed by an underscore (`\_`) signifies a nested command. In this example, the **c** command is nested within the **tog** command. For each additional nested command, the traffic pane indents the backslash-underscore pair further right.

The traffic pane only displays messages for the current MULTI session. Within one session, you can view all the messages that MULTI has printed—even if you have disconnected from your target. Scroll up to see the beginning of the message log. To clear the messages, right-click in the traffic pane and select **Clear Pane** from the shortcut menu.

For more information about the commands included in the preceding examples, see the *MULTI: Debugging Command Reference* book.

The Status Bar

The Debugger window status bar is located below the command pane, to the right of the pane-switching tabs.



The bar is divided into two areas. The box on the left displays informational messages. The following table lists the most common messages and their meanings.

Message	Meaning
In section: <i>section</i>	Displays the name of the program section where the PC pointer is currently located.
<i>item_description</i>	Explains the function of the button or field under your mouse pointer. For example, if you place your mouse pointer over the  button on the toolbar, the message <code>Run program from current position</code> appears in the status bar.
Srch: <i>string</i>	Indicates that the incremental search utility is searching the source pane for the pattern <i>string</i> . See “Incremental Searching” on page 130.

The box on the right displays process state information and debug server messages, such as those listed in the following table.

Process State Message	Meaning
CONTINUING	The program being debugged is preparing to resume execution.
DYING	The process being debugged is dying.
EXEC'ING	The process being debugged is performing an exec or is still executing startup code.
NO PROCESS	The process to be debugged has not started.
RUNNING	The process being debugged is running.
STOPPED	The process being debugged is stopped.
STOPPED INSIDE	The process being debugged is stopped inside an assembly instruction.
ZOMBIE	The process being debugged has exited, but data structures describing it still exist on the target.

Connecting to Your Target

Chapter 4

This Chapter Contains:

- Working with Connection Methods
- Standard Connection Methods
- Custom Connection Methods
- Temporary Connection Methods
- Simulator Connections
- Native Development Connections
- Freeze-Mode and Run-Mode Connections to the Same Target
- Using the Connection Organizer
- Disconnecting from Your Target

This chapter describes how to use the tools listed below to create, save, view, manage, and use Connection Methods. It also describes how to view information about your targets and manage them.

- **Connection Editor** — Allows you to create, save, and edit Connection Methods.
- **Connection Chooser** — Allows you to create or use Connection Methods to connect to your target.
- **Connection Organizer** — Allows you to manage all of your Connection Methods.

The next chapter, Chapter 5, “Preparing Your Target”, outlines the steps you must perform before you can run a program on the target you are connected to.

Working with Connection Methods

Before you can run your program, you must connect MULTI to your target. Configuring your debugging interface so that MULTI can connect to your target can be a complicated process. However, MULTI provides several graphical tools that simplify this process by allowing you to create as many *Connection Methods* as you need. A Connection Method contains a template that MULTI uses to connect to your target hardware or simulator. Whether you are using an on-chip debugging solution, an in-circuit emulator, a ROM monitor, an embedded RTOS, or a simulator to perform debugging, you need to create a Connection Method that contains all of the configuration settings required for connecting to your target.



Note In addition to configuring at least one Connection Method, you may also need to configure your target board and/or hardware debugging device (if applicable) before you can download and debug code. For information about setting up your target system and configuring the specific connection options available for your target, see the *MULTI: Configuring Connections* book for your target processor. For information about setting up and connecting to a Green Hills Probe or SuperTrace Probe, see the *Green Hills Debug Probes User's Guide*.

Tools Overview

The simplest way to create and edit Connection Methods is to use the **Connection Editor**, which allows you to make selections and enter settings through a graphical interface. The **Connection Editor** translates your selections and input into the appropriate command line options to the *debug server* that MULTI uses to connect to your specific target. Connection Methods created using the **Connection Editor** are referred to as *Standard Connection Methods*. Standard Connection Methods can be saved, invoked quickly using the **Connection Chooser**, and managed using the **Connection Organizer**. For more information about creating, configuring, and using Standard Connection Methods, see “Standard Connection Methods” on page 57.



Tip Users who prefer to enter command line options rather than use the GUI interface can create *Custom Connection Methods*. Like Standard Connection Methods, Custom Connection Methods can be saved, invoked quickly using the **Connection Chooser**, and managed using the **Connection Organizer**. For more information about using Custom Connection Methods, see “Custom Connection Methods” on page 67.

After you have created one or more Connection Methods, you can connect to your target from the **Connection Chooser** or **Connection Organizer**. You can use the **Connection Organizer** to save Connection Methods and organize them into Connection Files in your projects.

Standard Connection Methods

The following sections explain how to use MULTI to create, configure, save, edit, and use Standard Connection Methods.

Creating a Standard Connection Using the Project Wizard

If you use the **Project Wizard** to create a project, one or more Standard Connection Methods are created for you automatically. The new project contains a **default.con** Connection File that contains these Methods. (The **default.con** file is located in the **Target Resources** project in the Project Manager.) You may need to use the **Connection Editor** to edit the settings of automatically created Connection Methods before you can use them to connect to your target. For instructions about how to edit new or existing

Standard Connection Methods, see “Configuring a Standard Connection with the Connection Editor” on page 59.

Creating a Standard Connection Using the Connection Chooser

If you did not use the **Project Wizard** to create a project, you can create a new Standard Connection Method using the **Connection Chooser**. To do so, perform the following steps.

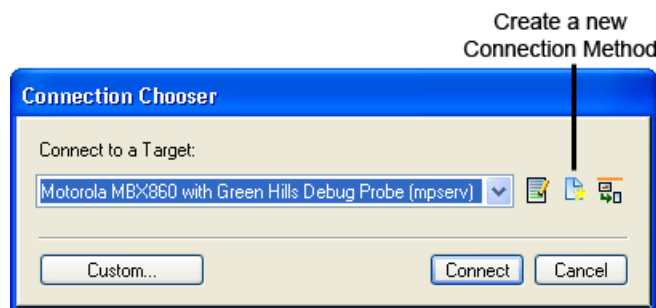
1. Open the **Connection Chooser**.

You can open the **Connection Chooser** from the MULTI Project Manager, Debugger, or Launcher. Perform one of the following steps to open the **Connection Chooser**:

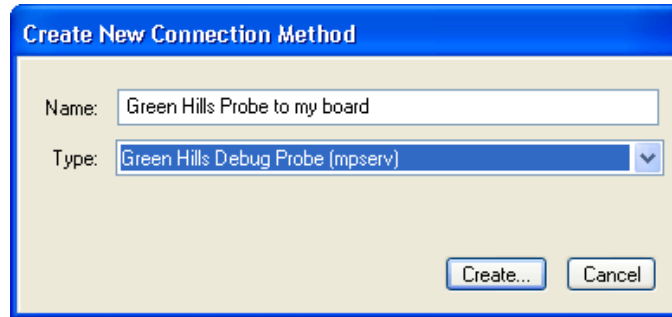
- From the MULTI Launcher, click  and select **Connect**, or select **Components** → **Connect**.
- From the MULTI Project Manager, click , or select **Connect** → **Connect**.
- From the MULTI Debugger, click , or select **Target** → **Connect**.



Note In a native environment, opening the **Connection Chooser** from the Debugger automatically connects to a native debug server. Instead, open the **Connection Editor** by selecting **Method** → **New** in the **Connection Organizer**.



2. Click to open the **Create New Connection Method** dialog box.



3. Enter a name for your Connection Method (for example, Green Hills Probe to my board). If you do not enter a name, the Method you create is a Temporary Connection Method (see page 69).
4. Select an appropriate connection type from the drop-down list. The types listed depend on your particular MULTI installation. Select the type that best describes your debug connectivity method.
5. Click **Create**. The Connection Method is stored in the [User Methods] file. For more information, see “The Default Connection File [User Methods]” on page 78.

When you click **Create**, a **Connection Editor** window opens for your new Standard Connection Method. You may need to edit the new Method before you use it for the first time. The next section describes how to do this.

Configuring a Standard Connection with the Connection Editor

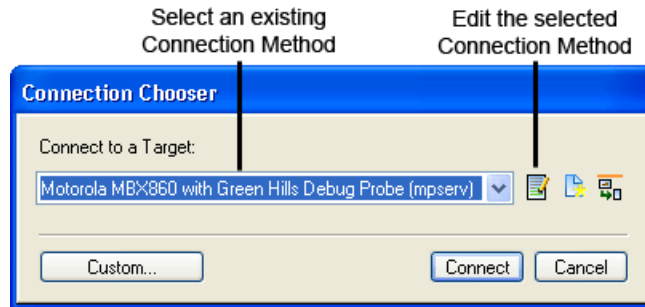
Before you can use a Standard Connection Method for the first time, you may need to configure it for your specific target system and debugging options. You can use the **Connection Editor** to configure a Standard Connection Method.

Opening the Connection Editor

If you create a new Standard Connection Method from the **Create New Connection Method** dialog box, the **Connection Editor** appears automatically when you click **Create**.

If you used the **Project Wizard** to create a project and you need to edit a default Connection Method created by the wizard, or if you want to edit a Standard Connection Method you have previously created and saved, perform the following steps to open the **Connection Editor**.

1. Open the **Connection Chooser** (see “Creating a Standard Connection Using the Connection Chooser” on page 58).

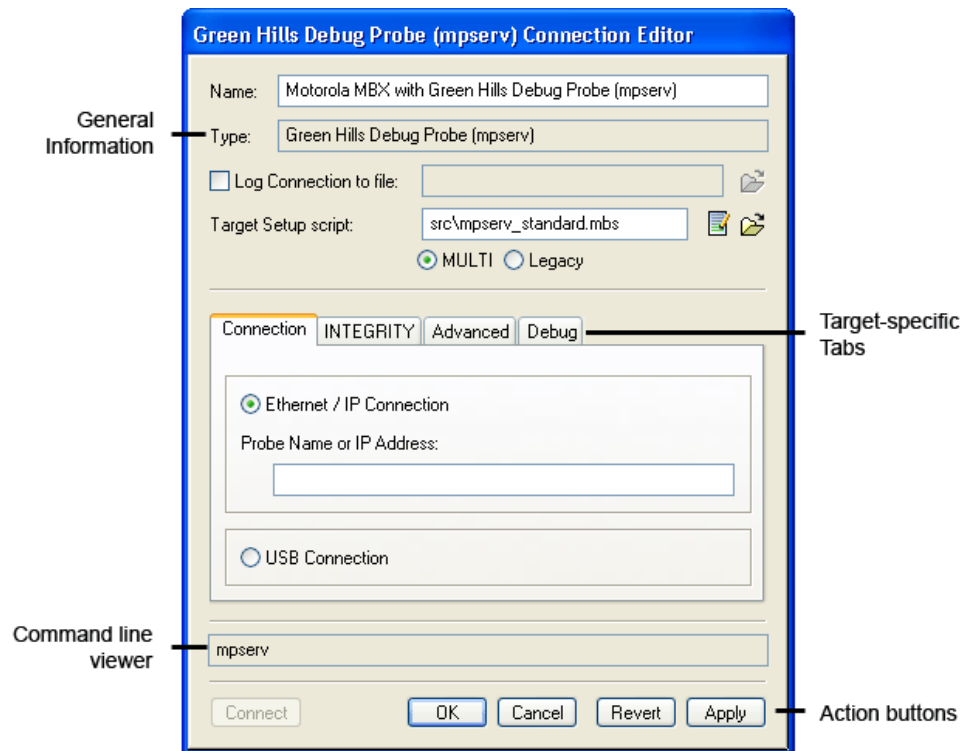


2. From the drop-down list, select the Connection Method you want to edit.
3. Click  to open the **Connection Editor** for the selected Connection Method.

For information about the basic features of the **Connection Editor**, see the next section.

The Connection Editor

The **Connection Editor** is target specific. You can use it to configure and edit Connection Methods.



The **Connection Editor** window for Standard Connection Methods has four sections, which are called out in the above graphic and documented in the following sections.

General Information Settings

The first section of the **Connection Editor** contains fields that display and/or set basic properties of a Connection Method. These fields are standard and appear on nearly all **Connection Editor** windows. The following table describes each field.

Name
Displays the name of the Standard Connection Method you define or edit. You can change the name of your method by editing this field.
Type
Displays the type of debugging interface the Connection Method is for, which determines which debug server MULTI uses. The connection type is specified when you create a Standard Connection Method. You cannot edit this read-only field.
Log Connection to file
Enables you to log transactions between MULTI and your debug server. To specify the file to write the log to, enter a pathname or click  to browse for the desired file. On UNIX, if you check the option box but do not specify a filename, MULTI writes the log to standard error output. Logging is disabled by default.

Target Setup script

Specifies your target setup script file. This is an optional field because not all targets require setup scripts.

If you require a setup script to configure your board, specify the filename (and full path, if necessary) of the appropriate script or click  to browse for it. When this Connection Method is used, MULTI usually runs the specified script before each download. If you are editing a default Connection Method created by the **Project Wizard** for your processor-board combination, the necessary setup script file (if applicable) appears in this field automatically.

To open the Editor on the target setup script, click . If no setup script is specified in this field, the Editor opens on a file with a filename derived from the debug connection's name.

For more information about target setup scripts, see the *MULTI: Configuring Connections* book for your processor type.

Scripting language radio buttons

Specify the scripting language so that MULTI knows how to interpret the specified target setup script, where:

- **MULTI** — Indicates that the specified target setup script was written with the MULTI scripting language. This is the default setting. For more information about MULTI scripting in general, see Chapter 2, “Using MULTI Scripts” in the *MULTI: Scripting* book.
- **Legacy** — Indicates that the specified target setup script was written with the debug server scripting language. For more information about debug server scripts, see Appendix A, “Green Hills Debug Server Command and Scripting Reference” in the *MULTI: Configuring Connections* book.

In previous versions of MULTI, target setup scripts had **.dbs** extensions and used the Green Hills debug server scripting conventions and associated debug server commands. Beginning with the v4.0 release, that scripting language was deprecated, and support for it will be discontinued in a future release. The default setup scripts created by the **Project Wizard** have **.mbs** extensions and use MULTI Debugger commands and scripting conventions instead. You must indicate which scripting language your setup script uses so that MULTI interprets it properly. When using the MULTI scripting language, you can still specify debug server commands by using the MULTI **target** command. For information about this command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.

For more detailed information about editing and using setup scripts, see Chapter 2, “Setting Up Your Target Hardware” in the *MULTI: Configuring Connections* book.

Target-Specific Tabs

The second section of the **Connection Editor** contains one or more tabs that contain fields for setting options specific to your debugging interface, type of connection, and target architecture. The number of tabs and the fields that appear vary according to the type of Standard Connection Method you are editing.

When the **Connection Editor** is first displayed after you create a new Connection Method, the settings and options are set to default values. Settings and options that are not available on your host operating system may appear dimmed. Some of the fields may require user input before the Connection Method can be used.

The following table describes in general terms the types of settings and options that appear on the most common tabs.

Tab	Description
Connection tab	The options on this tab usually require input for the Connection Method to function properly. The settings you choose for the connection options are probably different for each new target you connect to.
INTEGRITY tab	The option on this tab is useful if you are debugging an INTEGRITY system. For more information, see “Setting a Run-Mode Partner” on page 72.
Download tab	The options on this tab allow you to customize the process of downloading programs to the target. For example, this tab allows you to control what program sections are downloaded to the target.
Advanced tab	The options on this tab allow a high degree of control over the details of your connection. These options can often be left in their default states. Use this tab sparingly because changing the advanced options from their default settings may cause problems with your connection.
Debug tab	The options on this tab are provided to help troubleshoot your connection. You should not change these settings unless instructed to do so by Green Hills Technical Support.

Most **Connection Editors** include at least a **Connection** tab and a **Debug** tab, but many contain additional tabs. For most connections, you need to set or edit some of the fields on these tabs for your Standard Connection Method to work properly. For a complete description of the configuration settings available for

your target, see the *MULTI: Configuring Connections* book for your target processor. For more information about the **Connection Editor** settings for the Green Hills Probe and SuperTrace Probe, see “The Green Hills Debug Probe (mpserv) Connection Editor” in Chapter 2, “Connecting MULTI to Your Target” in the *Green Hills Debug Probes User’s Guide*.

The Command Line Viewer

The third section of the **Connection Editor** displays the command line equivalents of your GUI selections. Default selections do not always have corresponding commands or options on the command line.

The Command Line Viewer is a read-only field. You cannot make changes to the command line by typing in this field.

Changes you make in the GUI will not be reflected in the command line viewer until you click **Apply** or **OK**.

Action Buttons

The fourth section of the **Connection Editor** contains buttons that allow you to discard your edits, apply your edits, or connect using a Connection Method. The following table describes each button.

Button	Description
Connect	Records your changes, connects to your target using the Standard Connection Method you are editing, and closes the Connection Editor .
OK	Records your changes and closes the Connection Editor .
Cancel	Discards your changes and closes the Connection Editor .
Revert	Discards your changes and leaves the Connection Editor open.
Apply	Records your changes and leaves the Connection Editor open.



Note Clicking **Connect**, **OK**, or **Apply** in the **Connection Editor** saves your Standard Connection Method. In the future, the Connection Method will appear in the list of available Connection Methods in the **Connection Chooser** and the **Connection Organizer**.

For more information about using and managing existing Connection Methods, see “Using the Connection Organizer” on page 75.

Connecting with a Standard Connection

After you have created and configured a Connection Method, you can use it to connect to your target from the MULTI Launcher, the MULTI Project Manager, the MULTI Debugger, the **Connection Chooser**, the **Connection Editor**, or the **Connection Organizer**, as described in the following table.

From the	Perform These Steps
MULTI Launcher	Click the  button, and select the Connection Method you want to use.
MULTI Project Manager	Select Connect → Connect , or click  to open the Connection Chooser . Use the Connection Chooser to connect to your target. (See Connection Chooser below.)
MULTI Debugger	Select Target → Connect , or click  to open the Connection Chooser . Use the Connection Chooser to connect to your target. (See Connection Chooser below.) In a native environment, the Debugger automatically connects to a native debug server without opening the Connection Chooser .
Connection Chooser	Select the Connection Method from the drop-down list, and click Connect .
Connection Editor	Click the Connect button if it is available. If the Connect button appears dimmed, click OK and use the Connection Chooser to connect to your target. (See Connection Chooser above.)
Connection Organizer	Select a Connection Method. Then select Method → Connect to Target .

If your attempt to connect is unsuccessful, MULTI displays diagnostic information to help you understand the problem. The amount of the diagnostic information that MULTI can provide depends on the nature of your target. When you know what changes you need to make to your Connection Method, you can use the **Connection Editor**. For general information about using the **Connection Editor**, see “Configuring a Standard Connection with the Connection Editor” on page 59. For more information about the **Connection Editor** settings available for your particular target and for more information about diagnosing connection problems, see the *MULTI: Configuring*

Connections book for your target processor or, for Green Hills Probe or SuperTrace Probe users, the *Green Hills Debug Probes User's Guide*.

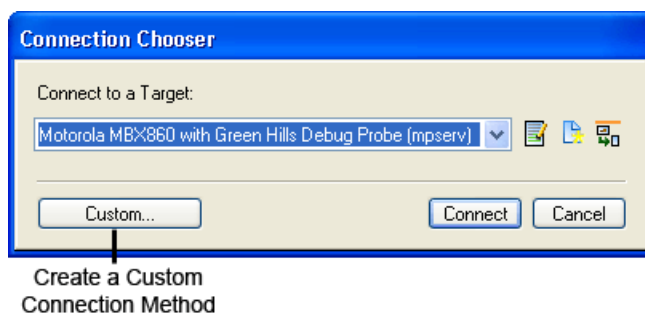
Custom Connection Methods

To create a Custom Connection Method, do one of the following:

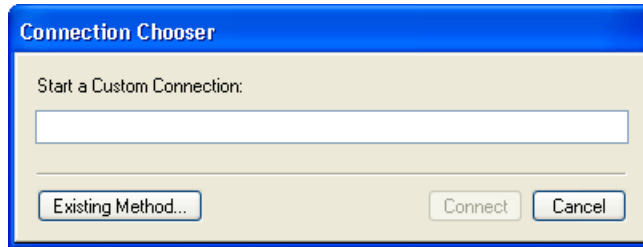
- Use the **connect** Debugger command. For information about this command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.
- Choose **Custom** as the connection type in the **Create New Connection Method** dialog box. For remaining steps about how to create a Custom Connection Method from the **Create New Connection Method** dialog box, see the instructions that follow the appearance of this dialog box in “Creating a Standard Connection Using the Connection Chooser” on page 58.
- Manually enter commands and options into the **Connection Chooser**. For more information, see below.

To create a Custom Connection Method from the **Connection Chooser**, perform the following steps:

1. Open the **Connection Chooser** (see “Creating a Standard Connection Using the Connection Chooser” on page 58).



2. Click **Custom**.



3. In the **Start a Custom Connection** text field, enter the connection command for your particular target connection.

The general format of the command that starts a debug server and connects to a target is:

`[setup=setup_script] xserv required_arguments... [options]...`

where:

- *setup_script* is the filename of the target setup script written in the MULTI scripting language. `setup=setup_script` may not be required for your target.
- *xserv* is the name of the Green Hills debug server that supports your debugging interface, monitor, or simulator. For example, **mpserv** is the Green Hills debug server that supports the Green Hills Probe. For the name of the correct debug server, see the *MULTI: Configuring Connections* book for your target processor.
- For information about the *required_arguments* and available *options* for your specific target, see the *MULTI: Configuring Connections* book for your target processor.

For more information about the Custom Connection Method settings for a Green Hills Probe or SuperTrace Probe, see the *Green Hills Debug Probes User's Guide*.

4. Click **Connect**.

MULTI connects to your target and saves your Custom Connection Method. In the future, the Method you entered in the text field will appear in the list of Connection Methods available in the **Connection Chooser** and **Connection Organizer**.

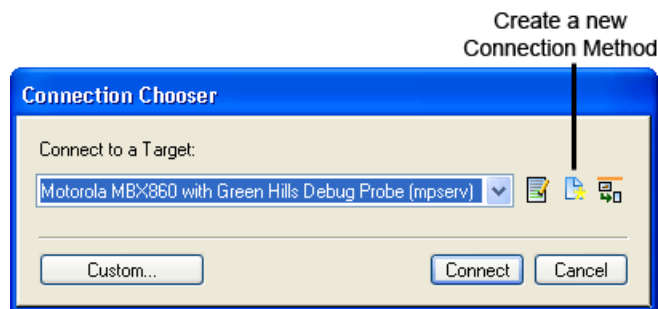
For more information about using and managing existing Connection Methods, see “Using the Connection Organizer” on page 75. To troubleshoot a connection problem, see the *MULTI: Configuring*

Connections book for your target processor or, for Green Hills Probe or SuperTrace Probe users, the *Green Hills Debug Probes User's Guide*.

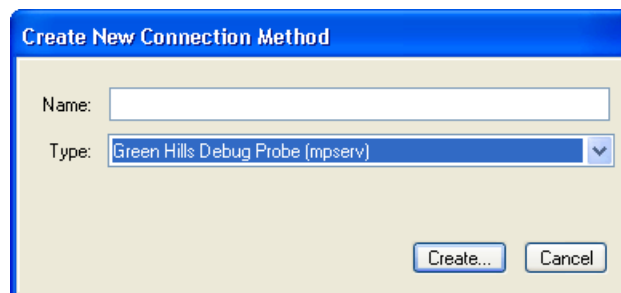
Temporary Connection Methods

You can create a connection that only exists for your current MULTI session, even if you save the file containing it, by creating a *Temporary Connection Method*. Temporary Connection Methods can be either Standard or Custom Connection Methods. To create a Temporary Connection Method, perform the following steps:

1. Open the **Connection Chooser** (see “Creating a Standard Connection Using the Connection Chooser” on page 58).



2. Click  to open the **Create New Connection Method** dialog box.
3. Leave the name field blank, and select the appropriate connection type from the drop-down list. To create a Custom Connection Method, select **Custom** from the drop-down list.



4. Click **Create**.
5. The new Connection Method is named Temporary Connection (#). Use the **Connection Editor** to configure the connection. For information about the **Connection Editor**, see “Configuring a Standard Connection with the Connection Editor” on page 59.

To convert a Temporary Connection Method to a permanent Standard or Custom Connection Method, open the **Connection Organizer**, right-click the Temporary Connection Method you want to change, and select **Make Permanent** from the shortcut menu. The method is saved and the name is changed to `Connection (#)`.

Simulator Connections

Even if your hardware is unavailable during development, you can still debug your program with the MULTI Debugger by connecting to a simulator for your target architecture type. A simulator is installed automatically when you install MULTI, and the procedure for connecting to it from the MULTI Debugger is similar to the procedure for connecting to an external target. In this chapter, the word *target* indicates either a hardware target or a simulated target. For more information about the simulator(s) available for your target, see the *MULTI: Configuring Connections* book for your specific processor.

Native Development Connections

If you are developing in a native environment, MULTI generally “connects” transparently through a simple debug server. Because this happens automatically and because configuration options cannot usually be set for such connections, native connections are not discussed in this book. If useful connection options are available for your particular native connection type, they are documented in the *MULTI: Building Applications* book for your specific native environment.

Freeze-Mode and Run-Mode Connections to the Same Target

The Debugger allows you to simultaneously debug the same target system via two different connections, one to the hardware (a freeze-mode connection) and one to the operating system (a run-mode connection). The following sections describe important features and limitations that you should be aware of when debugging a target in this way.

Automatically Established Run-Mode Connections

By default, the Debugger automatically establishes a run-mode target connection, called a *run-mode partner*, when you download and run an INTEGRITY kernel via a freeze-mode connection to a GHS simulator or to one of the GHS hardware debug solutions (Green Hills Probe or SuperTrace Probe). After INTEGRITY has booted, the run-mode partner is established in the same Debugger window as the freeze-mode connection.

Run-mode partnering puts certain measures in place to prevent run-mode connections from being closed prematurely. If you halt a freeze-mode connection, the run-mode partner becomes inaccessible: you cannot attach to or otherwise control tasks, nor can you read or write registers or memory. (But you may be able to continue browsing source code in tasks that you are already attached to.) These measures prevent the Debugger from erroneously attempting to communicate with the run-mode debug server while the target is halted via the freeze-mode connection.

If you kill, restart, or disconnect from a freeze-mode connection, the Debugger automatically disconnects from any run-mode partner set on the same target.



Note Run-mode partnering is only supported if you are debugging an INTEGRITY target and if you are connected to a GHS simulator or to one of the GHS hardware debug solutions (Green Hills Probe or SuperTrace Probe) via a freeze-mode connection. The run-mode partner requires an Ethernet connection to the target, and is not triggered by INTEGRITY until after the target has an IP address. (If you are using DHCP, the connection is not triggered if the board cannot talk to the DHCP server.)

The following limitations apply to run-mode partnering:

- Automatic establishment of the run-mode connection is aborted if you are stepping through kernel startup code or if any breakpoints are set on the freeze-mode connection when MULTI initializes the run-mode connection.
- If you are running a pre-INTEGRITY-10 kernel, the run-mode connection is only automatically established if the Idle Task gets to run—that is, if your system has at least some idle time and the processor is not being fully scheduled by the kernel.

See also “Troubleshooting” on page 74.

Setting a Run-Mode Partner

The **Set Run-Mode Partner** dialog box appears automatically the first time you download and run your INTEGRITY kernel using a freeze-mode Connection Method. Select an existing run-mode Connection Method in the drop-down list, or create a new run-mode Connection Method by clicking the **Create a new Connection Method** () button. The run-mode connection is automatically established when your INTEGRITY kernel boots.

To modify the run-mode partner setting for your freeze-mode Connection Method, perform one of the following actions:

- Select a freeze-mode connection in the target list, and choose **Target → Set Run-Mode Partner**, or enter **set_runmode_partner** in the command pane. (For information about the **set_runmode_partner** command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.) Select or create a run-mode Connection Method.
- Use the **Connection Editor** to edit the freeze-mode Connection Method. Select the **INTEGRITY** or **Connection** tab, and enter the name of an existing run-mode Connection Method in the text field labeled **Run-Mode Partner Connection**. For information about the **Connection Editor**, see “Configuring a Standard Connection with the Connection Editor” on page 59.

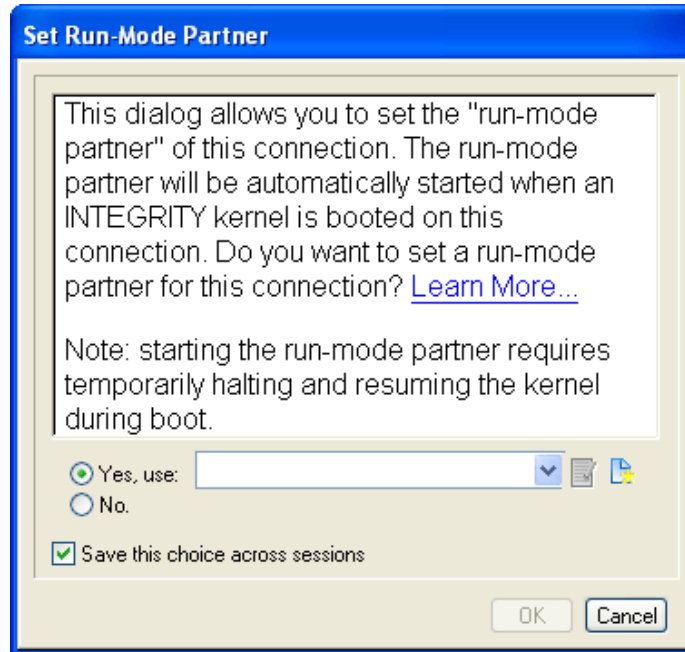
The run-mode connection you specify is automatically established when your INTEGRITY kernel boots.



Tip If you lose your run-mode connection, you can enter the command **connect -restart_runmode** in the Debugger command pane to try to reconnect. For more information, see the **connect** command in “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.

The Set Run-Mode Partner Dialog Box

The **Set Run-Mode Partner** dialog box allows you to configure the automatically established run-mode target connection for your current freeze-mode connection. For information about accessing the **Set Run-Mode Partner** dialog box, see “Setting a Run-Mode Partner” on page 72.



The **Set Run-Mode Partner** dialog box contains the following options:

- **Yes, use** — Specifies the run-mode Connection Method that the Debugger automatically attempts to establish when you boot an INTEGRITY kernel via the current freeze-mode connection. The drop-down list contains those Connection Methods known to be run-mode connections and, for versions 10 and later of INTEGRITY, an additional entry labeled **Automatically Constructed**. When you select **Automatically Constructed**, the operating system itself attempts to tell the Debugger what address and method to use to create a run-mode connection to the target.
- **No** — Disables the automatic establishment of a run-mode connection when you boot an INTEGRITY kernel via the current freeze-mode connection.
- **Save this choice across sessions** — If selected, the settings in the dialog box are associated with the current freeze-mode debug connection and used in future debug sessions. If cleared, the dialog box settings are used only until you exit MULTI, after which the freeze-mode connection reverts to whatever run-mode partner it had last (if any).

Disabling Automatically Established Run-Mode Connections

To disable run-mode partnering for a particular freeze-mode connection, perform the following steps:

1. In the target list, select the freeze-mode connection.
2. Select **Target** → **Set Run-Mode Partner**.
3. In the **Set Run-Mode Partner** dialog box, select **No**.
4. Optionally select **Save this choice across sessions**.
5. Click **OK**.

Alternatively, you can select the freeze-mode connection in the target list and enter `set_runmode_partner -none` in the Debugger command pane. For information about the `set_runmode_partner` command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.

Troubleshooting

You may encounter the following problems while debugging via simultaneous freeze-mode and run-mode connections to the same target, regardless of whether or not the run-mode connection was created automatically via the run-mode partner functionality, or whether or not the freeze-mode and run-mode connections were created within the same Debugger process.

- Host I/O calls made on a freeze-mode connection can cause a run-mode connection to the same target to become unresponsive for the duration of the host I/O call. Note that some host I/O calls (for example, large block reads from, or writes to, host files) can take a long time for the Debugger to process. Also note that some host I/O calls, such as reads from standard input, may block indefinitely, waiting for you to type input in the I/O pane.
- Breakpoints that are set on a freeze-mode connection can cause the run-mode connection to the same target to become unresponsive until the freeze-mode connection is resumed. This applies regardless of whether hitting the breakpoint leaves the target halted, as in the case of a regular software breakpoint, or whether it resumes the target, as in the case of a conditional breakpoint whose condition is false.
- A single instance of the MULTI Debugger can deadlock between a freeze-mode and run-mode connection to the same target, even though

various safeguards have been introduced to prevent this problem in common cases.

If you encounter this problem, after a short delay (configurable via MULTI's `SERVERTIMEOUT` system variable), you may see a dialog box saying **Server message timed out. Terminate Connection?** with buttons labeled **Continue** and **Terminate**. To regain control of the Debugger and freeze-mode connection, click **Terminate**. You may also be able to re-initiate the run-mode connection request after this.

The following general information is intended to help you work around the preceding problems if you encounter them.

- For best results, take all the following precautions if you plan to leave a freeze-mode connection active while debugging using a run-mode connection to the same target:
 - Do not write code that makes freeze-mode host I/O calls.
 - Remove all freeze-mode breakpoints before initiating the run-mode target connection.
 - Launch a separate instance of the MULTI Debugger to connect to the target via the run-mode Connection Method.
- Using the run-mode partner functionality is preferable to manually establishing a run-mode connection. However, if you must manually establish a run-mode connection, do so several seconds after booting your kernel via the freeze-mode connection. For example, wait for the INTEGRITY kernel banner to appear on your target's serial port before connecting via `rtserv`.

Using the Connection Organizer

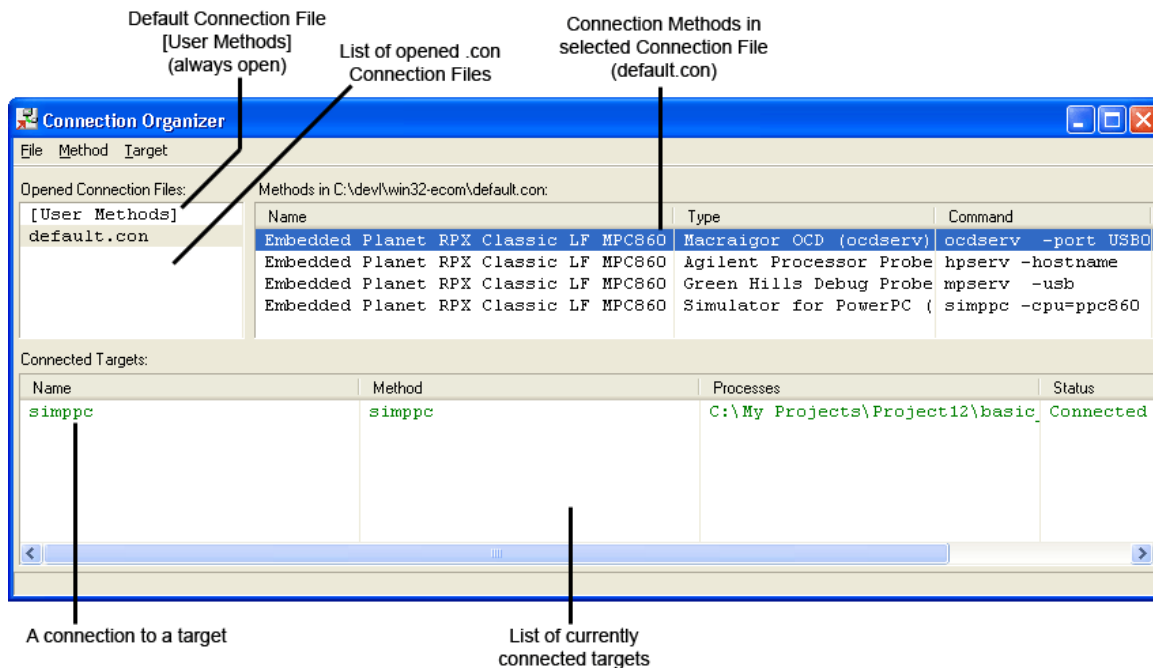
The **Connection Organizer** allows you to create, edit, copy, load, and save Connection Methods and connect to your target using Connection Methods. The following sections describe how to use the various features of the **Connection Organizer**. For more information about Connection Methods, see “Working with Connection Methods” on page 56.

Opening the Connection Organizer

You can open the **Connection Organizer** in any of the following ways:

- From the Launcher, click  and select **Open Connection Organizer**, or select **Components** → **Open Connection Organizer**.
- From the Project Manager, select **Connect** → **Connection Organizer**.
- From the Debugger, select **Target** → **Show Connection Organizer**.
- From the Debugger command pane, enter the **connectionview** command. For information about the **connectionview** command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.
- From the **Connection Chooser** dialog box, click . If the **Connection Chooser** dialog box automatically appeared as the result of an action you took (such as trying to run your program without first connecting), clicking the  button aborts the action and opens the **Connection Organizer**.

A sample **Connection Organizer** follows.



The **Connection Organizer** has three sections:

- **Opened Connection Files** — Lists all the Connection Files currently open. Each Connection File contains one or more Connection Methods. You can use Connection Files to save, restore, and transport your connection configurations. The **Connection Organizer** always has at least one Connection File open: the **User Methods** file. The **Opened Connection Files** list may also contain **default.con** files created by the **Project Wizard**, or it may contain other **.con** files you have created. For more detailed information about Connection Files, see “Creating and Managing Connection Files” on page 78.
- **Methods in selected file** — Lists all the Connection Methods present in the Connection File that is selected in the **Opened Connection Files** list. You can use the **Connection Organizer**’s menu choices and shortcuts to start, copy, edit, move, or delete Connection Methods from this list.
- **Connected Targets** — Lists all the currently established target hardware or simulator connections. For more detailed information about connected targets, see “Managing Your Connected Targets” on page 80.

Creating a Connection Method

You can create a new Connection Method from the **Connection Organizer**. To do this, select **Method** → **New** from the menu bar. The **Create New Connection Method** dialog box appears. For remaining steps, see the instructions that follow the appearance of this dialog box in “Creating a Standard Connection Using the Connection Chooser” on page 58. The Connection Method is stored in the file selected in the **Connection Organizer**’s **Opened Connection Files** list.

Editing a Connection Method

To edit a previously saved Connection Method from the **Connection Organizer**:

1. From the **Opened Connection Files** list, select the Connection File that contains your desired Connection Method.
2. From the **Methods in selected file** list, select a Connection Method.
3. From the **Connection Organizer**’s menu, select **Method** → **Edit** or right-click the selected method and select **Edit** from the shortcut menu.

4. Using the **Connection Editor** that appears, modify your Connection Method settings. For general information about using the **Connection Editor**, see “The Connection Editor” on page 61. For detailed information about the settings available for your particular target, see the *MULTI: Configuring Connections* book for your target processor or, for Green Hills Probe or SuperTrace Probe users, the *Green Hills Debug Probes User’s Guide*.
5. Click **OK** to save your changes.

Creating and Managing Connection Files

You can save one or more Connection Methods in a Connection File, which is a regular text file that usually ends with a **.con** extension. The Connection Methods stored in a Connection File are self-contained and portable. As a result, you can open and use Connection Files created by other MULTI installations. You can even email Connection Files to other users and computers.

You can open any number of Connection Files in the **Connection Organizer**. Some are opened for you automatically. The default Connection File, **[User Methods]** is always open. If the **Connection Chooser** appears, the **Connection Organizer** also opens the Connection File that contains the Connection Method selected by default in the **Connection Chooser**.

To view the Connection Methods stored in an open Connection File, select the Connection File in the **Opened Connection Files** list.

By default, changes to open Connection Files are saved immediately. If you change the default setting, you must manually save your Connection Files via the **Connection Organizer**. For information about changing the default settings, see the **autoSaveConnectionsInFiles** and **autoSaveUserConnections** options in the “Session Configuration Options” in Chapter 9, “Configuration Options” in the *MULTI: Editing Files and Configuring the IDE* book.

The Default Connection File [User Methods]

The first file in the **Opened Connection Files** list is always the **[User Methods]** file. It cannot be closed or renamed. This file serves as a default location for new Connection Methods when you do not explicitly create a Connection File for them. The contents of **[User Methods]** are stored in your Green Hills user directory as **multiconnections.con**. By default, changes to **[User**

Methods] are immediately saved. This behavior is configured separately from automatically saving other Connection Files. For more information, see the **autoSaveUserConnections** option in “Session Configuration Options” in Chapter 9, “Configuration Options” in the *MULTI: Editing Files and Configuring the IDE* book.

Connection Files in a Project

You can use the MULTI Project Manager to add Connection Files with the **.con** extension to your **.gpj** projects. By default, the **Connection Organizer** automatically opens any Connection Files in a project when the Project Manager opens that project. You can select the Connection File from the **Opened Connection Files** list and work with the Connection Methods that it contains.

To add a Connection File to a project:

1. Open the Project Manager, select **File → Open Project**. Select the filename of the project you want to work with.
2. Select **Edit → Add File into project.gpj**. Select the filename of the Connection File you want associated with this project. You can specify a Connection File that does not exist; the **Connection Organizer** simply creates it for you.

Connecting from the Connection Organizer

To connect to your target, you can use any appropriate Connection Method listed in the **Connection Organizer**. To connect from the **Connection Organizer**:

1. From the **Opened Connection Files** list, select the Connection File that contains your desired Connection Method.
2. From the **Methods in selected file** list, select a Connection Method.
3. From the **Connection Organizer** menu bar, select **Method → Connect to Target**, or right-click the method and select **Connect to Target** from the shortcut menu.
4. If the connection is successful, a new connection appears in the **Connected Targets** list. For information about connected targets, see “Managing Your Connected Targets” on page 80.

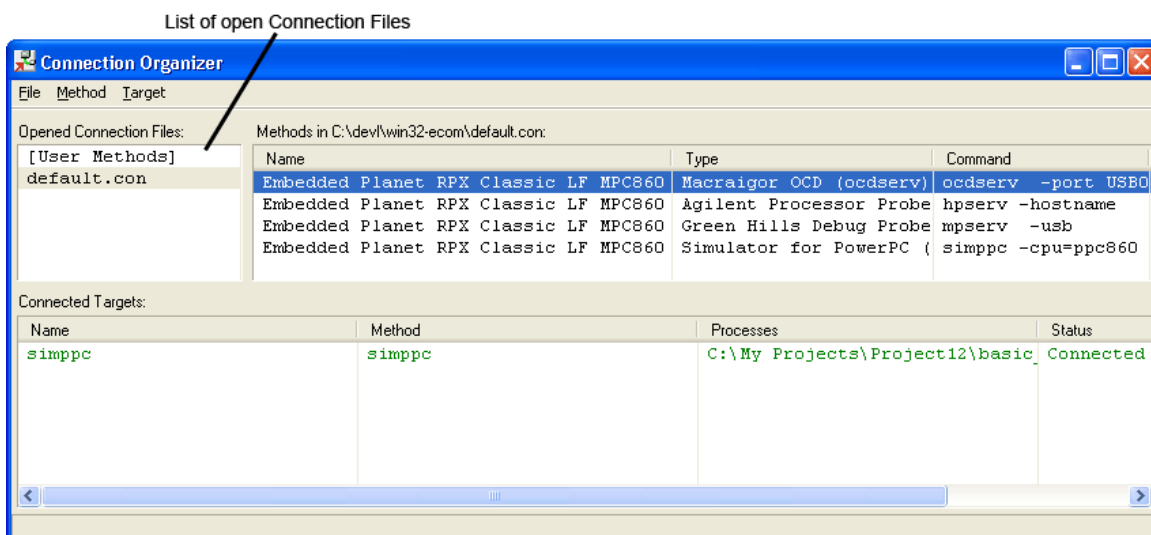
To troubleshoot a connection problem, see the *MULTI: Configuring Connections* book for your target processor or, for Green Hills Probe or SuperTrace Probe users, the *Green Hills Debug Probes User's Guide*.

Managing Your Connected Targets

After you have connected using a Connection Method, the connected target appears in the **Connected Targets** list. The **Connected Targets** list is located at the bottom of the **Connection Organizer**. From the **Connected Targets** list, you can find information about the target and control the target.

Connection Organizer Menu and Action Reference

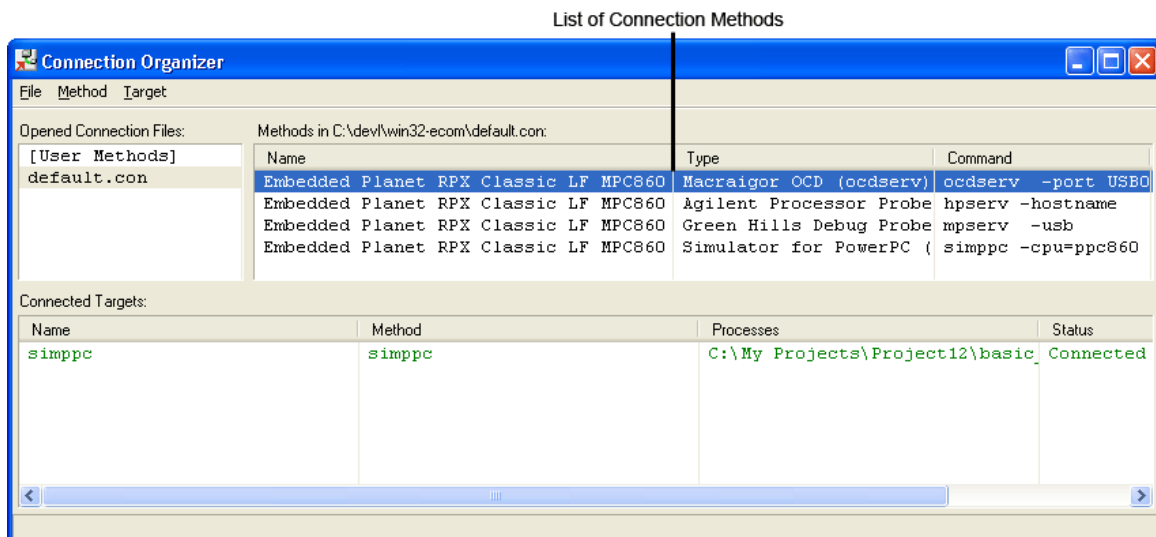
The primary interface for working with Connection Files is the list of **Opened Connection Files** located in the **Connection Organizer**.



You can select Connection Files from this list and perform the following actions by using the **File** menu. Most of the **File** menu options are also available via the right-click menu.

Menu Item	Meaning
New	Opens a dialog box that prompts you to name a new Connection File. When you click the Create button, the Connection File is created and added to the Opened Connection Files list.
Open	Opens a dialog box that prompts you to open an existing .con file. After you open it, the file appears in the Opened Connection Files list, and you can work with the Connection Methods contained in it.
Close	Closes the selected Connection File. Note that the [User Methods] Connection File is always open and cannot be closed.
Save	Saves any changes to the selected Connection File.
Save a copy	Opens a dialog box that prompts you to choose a different filename for the selected Connection File before saving it.
Save all	Saves all changes to all open Connection Files.
Close Window	Closes the window. This menu item does not affect the Connection File.

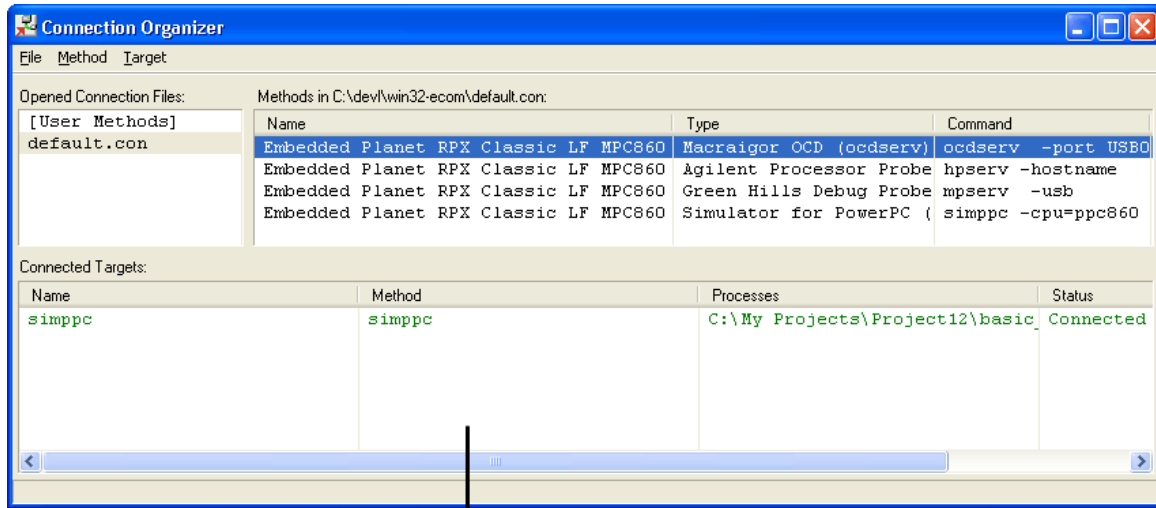
The primary interface for working with Connection Methods is the list of Connection Methods located under **Methods in selected file** in the **Connection Organizer**.



You can select Connection Methods from this list and perform the following actions by using the **Method** menu or the right-click menu.

Menu Item	Meaning
New	Opens a dialog box that prompts you to define a new Connection Method with a name, a type, and options that you specify.
Connect to Target	Uses the settings of the selected Connection Method to connect to the target. If the connection is successful, the connected target appears in the Connected Targets list. If the connection is unsuccessful, an error message and diagnostic information appear.
Edit	Opens a Connection Editor that allows you to change the settings of the selected Connection Method. See “The Connection Editor” on page 61.
Copy	Opens a dialog box that allows you to copy the selected Connection Method into either the same or a different Connection File. If a Connection Method with the same name already exists in the Connection File you copy to, the copy you create is named <i>method_name</i> (2) or <i>method_name</i> (3), etc.
Move	Opens a dialog box that allows you to move the selected Connection Method into another Connection File. If a Connection Method with the same name already exists in the Connection File you move the selected Connection Method into, the Method you move is renamed <i>method_name</i> (2) or <i>method_name</i> (3), etc.
Delete	Deletes the Connection Method selected in Methods in selected file from the current Connection File.

The primary interface for working with connected targets is the list of **Connected Targets** located in the **Connection Organizer**.



Connected targets list

You can select connected targets from this list and perform the following actions by using the **Target** menu. Most of the following **Target** menu options are also available via the right-click menu.

Menu Item	Meaning
Show Task Manager	Displays the Task Manager associated with the selected target. This option is only available if the target supports multiple tasks. For more information about the Task Manager, see “The Task Manager” on page 424.
Set Logging	Opens the Target Logging Settings dialog box, which allows you to capture communications between MULTI and the target’s debug agent.
Disconnect	Disconnects the selected target from MULTI. Processes that are running or being debugged on the target may be halted or may continue running undisturbed. The exact behavior of Disconnect is dependent on your target and connection. For example, when you disconnect from a simulator, the simulator exits, ending all processes it is running. However, an RTOS debug agent detaches and allows the underlying RTOS to continue running.

Disconnecting from Your Target

To disconnect from your target, do one of the following:

- In the Debugger command pane, enter the **disconnect** command. For information about the **disconnect** command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.
- In the target list, select a connected executable. Click the **Disconnect** button () or select **Target → Disconnect from Target**.
- In the target list, right-click a connected executable and then select **Disconnect from Target** from the shortcut menu that appears.
- In the **Connection Organizer**, select a connection from the **Connected Targets** list and then select **Target → Disconnect**.
- In the **Connection Organizer**, right-click a connection in the **Connected Targets** list and then select **Disconnect** from the shortcut menu that appears.

Preparing Your Target

Chapter 5

This Chapter Contains:

- Chapter Terminology
- Associating Your Executable with a Connection
- Preparing Your Target
- Related Settings

Before you can run or debug a program on your target, you must download it to the target's RAM, program it into the target's flash memory, or verify through the Debugger that it is already present on the target. This chapter describes how you can prepare your target to be debugged using one of these methods.

Chapter Terminology

This chapter uses terms that have specific meanings in the following sections. These terms are defined in the list below:

- **Executable** — Any executable; thread; or INTEGRITY application, module, or kernel that you can select from the target list for debugging. This definition only applies to this chapter.
- **CPU** — An entity that an executable runs on. When debugging in freeze mode, this is an actual CPU. When debugging in run mode, it is the CPU abstraction provided by the operating system, which in some cases corresponds to more than one actual CPU.
- **Downloading** — Writing the executable into RAM on your target.
- **Flashing** — Writing the executable into flash memory on your target.
- **Verifying** — Reading memory (either RAM or flash) from your target and comparing it to the contents of the executable, thus ensuring that the executable loaded into the Debugger is the same as the executable loaded onto your target.

For information about phrases that are used in the target list, see “Target Terminology” on page 36. For information about statuses that appear in the target list, see “The Status Column” on page 37.

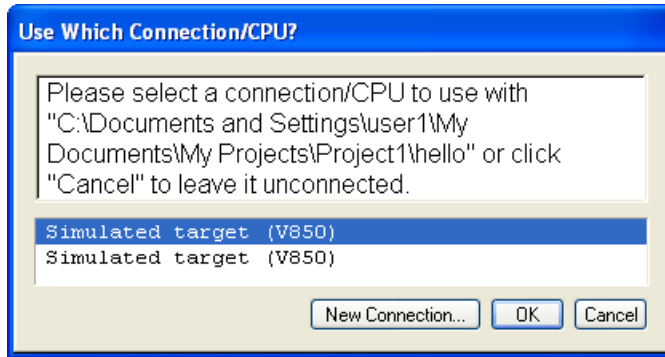
Associating Your Executable with a Connection

When you open an executable in the Debugger, the Debugger automatically associates it with a connection if one is available and applicable to the executable. Otherwise, the executable is not associated with any connection and appears in the target list under the heading **Unconnected Executables**. Before you are able to download, flash, or verify the executable, you must establish a connection to your target and associate the executable with the connection.

To do so, select the executable in the target list and then perform one of the following actions.

- Click the **Connect** button () . Using the **Connection Chooser** that appears, connect to a target that contains a CPU compatible with the executable.
- Select **Debug → Use Connection**. The submenu that appears lists compatible, currently active connections. If a connection appears dimmed, the current executable cannot be associated with that connection. Select an **Available** connection (if any) from the top of the menu, or select **Create New Connection**. (**Available** indicates that the connection can accept more executables. **Current** indicates that the connection is associated with the current executable. **Full** indicates that using the connection will cause another executable to stop using it. See “The Use Connection Submenu” on page 509.)
- In the Debugger command pane, enter the **change_binding bind** command. The **Connection Chooser** prompts you to establish a connection. For more information, see the **change_binding** command in “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.

After you perform one of the preceding operations, the executable is automatically associated with the connection if only one compatible CPU exists on the target you connected to. (This is usually the case.) If the selected executable is compatible with more than one CPU on the target, the **Use Which Connection/CPU?** dialog box appears. The following graphic is an example of the **Use Which Connection/CPU?** dialog box for a multi-core V850 simulator.



Select the connection that contains the correct CPU.

To disassociate the executable from the connection, select the executable and then perform one of the following actions:

- Click the **Disconnect** button () to disconnect from the target.
- Select **Debug** → **Use Connection** → **Stop Using Current Connection**.
- In the Debugger command pane, enter **change_binding unbind**. For more information, see the **change_binding** command in “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.



Note All the menu items listed in this section are also accessible from the shortcut menu that appears when you right-click an executable.

For more information about connecting, see Chapter 4, “Connecting to Your Target”. For information about how items are arranged in the target list after the executable is associated with the connection, see “The Target List Display” on page 35. For information about loading the executable on the target, see “Preparing Your Target” on page 90.

Updating Your MULTI 4 Target Connections

Specifying a download, attach, or board setup connection mode for your target connection, as was required in MULTI 4, is not supported in MULTI 5. This section summarizes how to obtain results similar to those of:

- Selecting **Download**, **Attach**, or **Board Setup** in MULTI 4’s **Connection Editor** or **Connection Chooser**

- Specifying `mode=download`, `mode=attach`, or `mode=boardsetup` in a MULTI 4 Custom Connection Method, **connect** command, or **-connect** command line option.



Tip To remove the deprecated `mode=setting` argument from a MULTI 4 Connection Method, edit and save the connection in MULTI 5.

Download Mode (`mode=download`)

Downloading your program in MULTI 5 works in roughly the same way as in MULTI 4. You can open a Debugger window on your executable, connect to your target, and download your program to your target just as you could before. In MULTI 5, you do not have to specify that your connection is a download mode connection.

Attach Mode (`mode=attach`)

After connecting to your target, select **<Direct hardware access>** from the target list to get run control of your target, to read and write registers and memory, and to see the raw disassembly view of whatever your target is running (as on a newly opened MULTI 4 attach mode connection with no executable). You no longer have to specify that your connection is an attach mode connection.

If an executable that you would like to debug is already loaded onto your target, open your executable in the Debugger and do one of the following:

- Select **Debug → Prepare Target**, and specify **Program already present on target. Verify: Not at all**.
- Run the Debugger command **prepare_target -verify=none** in the Debugger command pane. For information about the **prepare_target** command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.

Performing either of the preceding operations allows MULTI to assume that the program loaded on your target is the same as the executable you opened. In the target list, the **<Direct hardware access>** entry and your executable will merge together, allowing you to run, halt, and/or step your target with reference to your executable’s source code in the source pane.

Board Setup Mode (mode=boardsetup)

Because MULTI 4's board setup mode is an extension of attach mode, you can connect to your target without an executable, and click **<Direct hardware access>** as described in "Attach Mode (mode=attach)" on page 89. To get some of the other effects of the deprecated board setup mode, try enabling no stack trace mode and memory sensitive mode and disabling automatic coherency checking—settings for all of which appear under **Debug → Debug Settings**.

If you are connected via a Green Hills Probe, you can also disallow access to memory (or specific areas thereof) with the **ma** command. For information about the **ma** command, see Appendix A, "Command Reference" in the *Green Hills Debug Probes User's Guide*.

Preparing Your Target

Before you can run or debug a program on your target, you must download it to the target's RAM, program it into the target's flash memory, or verify through the Debugger that it is already present on the target. To do so, select an executable in the target list and then perform one of the following operations:

- Perform a run-control operation such as stepping or running your program.
- Click the **Prepare Target** button ().
- Select the **Prepare Target** menu item from the **Debug** menu or the right-click menu.
- In the Debugger command pane, enter the **prepare_target** command. For more information and options to this command, see "General Target Connection Commands" in Chapter 18, "Target Connection Command Reference" in the *MULTI: Debugging Command Reference* book.



Note If you have not connected to a target, the **Connection Chooser** prompts you to connect. If the selected executable is not automatically associated with the connection, the **Use Which Connection/CPU?** dialog box prompts you to pick a connection. For more information, see "Associating Your Executable with a Connection" on page 87.

After you perform one of the preceding operations, MULTI either opens the **Prepare Target** dialog box, which allows you to choose whether to download, flash, or verify the executable, or MULTI prepares your target for you by

automatically executing one of these actions. For information about how MULTI determines which action to execute automatically, see “Program Types” on page 92.

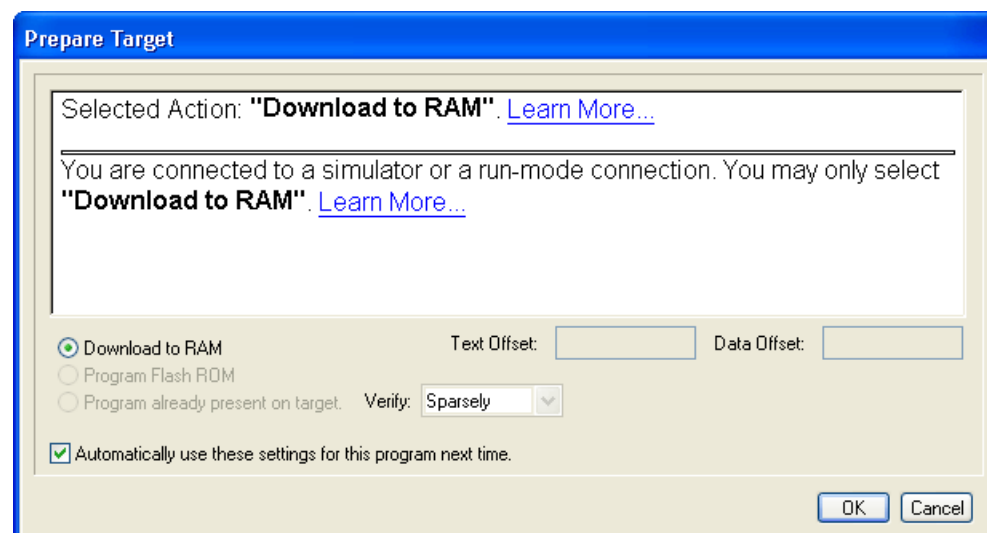
If you want to specify which action is performed (that is, you do not want MULTI to automatically download, flash, or verify the executable), select the **Prepare Target** menu item from the **Debug** menu or the right-click menu, or pass the **-ask** option to the **prepare_target** command. If you prepare the target by another means and at least one of the following items is true, MULTI automatically executes an action:

- Only one action is appropriate and MULTI does not require input (such as the text offset of a PIC program).
- The last time you prepared the target for the selected program, the option **Automatically use these settings for this program next time** was selected in the **Prepare Target** dialog box, and the program’s memory layout has not changed since then. (This option is selected by default.)

For more information about the **Prepare Target** dialog box, see the next section.

The Prepare Target Dialog Box

The **Prepare Target** dialog box contains options for specifying that MULTI downloads the executable, flashes the executable, or verifies the presence of the executable on the target.



Each option in the **Prepare Target** dialog box is listed below:

- **Download to RAM** — Writes the executable into RAM on your target. For more information, see “Downloading Your Executable” on page 95.
 - **Text Offset** — Allows you to specify where the text section will be located when the executable is downloaded. This option is only available for programs using position-independent code. For more information, see “Downloading Your Executable” on page 95.
 - **Data Offset** — Allows you to specify where the data section will be located when the executable is downloaded. This option is only available for programs using position-independent data. For more information, see “Downloading Your Executable” on page 95.
- **Program already present on target. Verify** — Specifies that the executable is already present in your target’s memory. Depending on the **Verify** option that you choose from the drop-down menu, MULTI may check to ensure that the executable is on your target. For more information, see “Verifying the Presence of Your Executable” on page 95.
- **Automatically use these settings for this program next time** — Remembers the **Prepare Target** dialog box settings for this executable. If the settings remain applicable, MULTI uses them automatically and does not open the **Prepare Target** dialog box unless you request otherwise. For more information, see “Preparing Your Target” on page 90.

The options that are available in the **Prepare Target** dialog box vary according to the item you are debugging. For more information, see the next section.

When you are finished making your selections, click **OK**. MULTI prepares your target.

Program Types

MULTI understands a number of general program types, and is able to determine the action (download, flash, or verify) that should either be selected by default in the **Prepare Target** dialog box or automatically executed (for more information, see “Preparing Your Target” on page 90). The following sections provide specific information about how MULTI determines a program’s type, what each type means, and what action MULTI defaults to for each type.



Note If you are using an instruction set simulator, MULTI may determine that **Download to RAM** is the only action possible, regardless of what program type is detected. In this situation, all other options in the **Prepare Target** dialog box are disabled, and even programs built to load from ROM (that is, ROM run and ROM copy programs) should be “downloaded.”

RAM Download Programs

If MULTI is unable to locate certain special symbols that indicate the program is ROM run or ROM copy (see the following sections), it determines that the program is a RAM download program. In this case, MULTI expects that all sections marked as allocated in the ELF file should be present on the target.

The **Prepare Target** dialog box defaults to **Download to RAM** for programs of this type. No other option is generally applicable, though if the program has already been downloaded to the target, it may be possible to verify that this is the case instead of re-downloading. If you are debugging a native target or a Dynamic Download INTEGRITY application, downloading is the only action possible. For information about downloading, see “Downloading Your Executable” on page 95. For information about verifying, see “Verifying the Presence of Your Executable” on page 95.

Only RAM download programs can have position-independent code or data, and if MULTI determines that either or both of those is present, the **Text Offset** and/or **Data Offset** text fields are available for input and must be filled in. For more information, see “Downloading Your Executable” on page 95.

ROM Run Programs

A ROM run program is stored to ROM and runs solely out of ROM (using RAM only for stack and heap space).

MULTI determines that a given program has this type if the special symbols `__ghs_rombootcodestart` and `__ghs_rombootcodeend` exist, and the symbols `__ghs_rambootcodestart` and `__ghs_rambootcodeend` do not exist. (Note that MULTI also searches for and uses various other special symbols, such as `__ghs_romstart`, `__ghs_romend`, `__ghs_ramstart`, and `__ghs_ramend`, to improve the debugging experience.)

If a program is of type ROM run, MULTI expects that the sections marked as allocated in the program's ELF file should be loaded into the target's ROM. This is generally achieved by programming the flash ROM on the target.

The **Prepare Target** dialog box defaults to **Program Flash ROM** for programs of this type. Any of the **Verify** options may also be appropriate if the program has already been programmed into the target's flash ROM. Downloading is generally not applicable, unless the program is being run on a simulator (see "Program Types" on page 92). For information about verifying, see "Verifying the Presence of Your Executable" on page 95.

ROM Copy Programs

ROM copy programs are initially located in ROM, but copy themselves to RAM during startup, and then execute partially or completely from RAM.

MULTI determines that a given program has this type if the linker-inserted symbols `__ghs_rombootcodestart`, `__ghs_rombootcodeend`, `__ghs_rambootcodestart`, and `__ghs_rambootcodeend` exist. (Note that MULTI also searches for and uses various other special symbols, such as `__ghs_romstart`, `__ghs_romend`, `__ghs_ramstart`, and `__ghs_ramend`, to improve the debugging experience.) If the special symbol `__ghs_after_romcopy` exists, MULTI is able to restore software breakpoints in the RAM image of the program after the ROM copy is completed.

If a program is of type ROM copy, MULTI expects that the sections marked as allocated in the program's ELF file should be loaded into the target's ROM. This is generally achieved by programming the flash ROM on the target.

The **Prepare Target** dialog box defaults to **Program Flash ROM** for programs of this type. Any of the **Verify** options may also be appropriate if the program has already been programmed into the target's flash ROM. Downloading is generally not applicable, unless the program is being run on a simulator (see "Program Types" on page 92). For information about verifying, see "Verifying the Presence of Your Executable" on page 95.

Unknown Programs

If MULTI is unable to determine anything about a program, the default action is **Download to RAM**. For information about downloading, see "Downloading Your Executable" on page 95.

When MULTI cannot detect any program information, it is generally the case that something is wrong with the program or its debug information.

Downloading Your Executable

To download your executable to your target's memory, select **Download to RAM** in the **Prepare Target** dialog box. This is the most commonly selected option for debugging embedded programs with MULTI.

If a board setup script is associated with the target connection, it is executed immediately before the executable is downloaded.

When the **Download to RAM** action is available and MULTI detects that your program has been linked with position-independent code and/or data, the **Text Offset** and/or **Data Offset** fields become available. These fields set the `_TEXT` and `_DATA` system variables, which allow you to specify where the text and data sections will be located when they are downloaded. If MULTI fails to detect that a program contains position-independent code or data, you can manually set the offsets using `_TEXT` and `_DATA`. For more information, see "System Variables" on page 291.

Verifying the Presence of Your Executable

To specify that your executable is already present in the target's memory, select **Program already present on target. Verify**. Depending on the **Verify** option you choose from the drop-down menu, MULTI may check to ensure that the contents of target memory match the contents of the executable program file.

The following list describes the **Verify** options:

- **Sparsely** — Verifies a few bytes at the beginning, middle, and end of all the downloaded sections that cannot be written to. The `.text` section is an example of one such section. Because certain sections of memory, such as `.bss`, `.data`, and `.heap`, may be written to during program execution, you can expect them to differ from the executable program file. When you specify this option, MULTI does not check these sections.

This option halts your target if it is running.

- **Completely** — Verifies (in entirety) all downloaded sections that cannot be written to. This may take a long time.

This option halts your target if it is running.

For information about downloaded sections, see the preceding bullet point.

- **Not at all** — Assumes, but does not verify that the contents of target memory match the contents of the executable program file. This option does not halt your target.

Related Settings

Memory Sensitive Mode

To enable memory sensitive mode in the Debugger, select **Debug** → **Debug Settings** → **Memory Sensitive**. You can also select this mode by setting the system variable `$_VOLATILE=1`. For more information about system variables, see “System Variables” on page 291.

Memory sensitive mode is used when debugging targets whose memory can change in ways unexpected to the Debugger or while examining memory-mapped I/O to avoid unexpected memory references. Some ways that a target could cause such unexpected changes are self-modifying code and references to dual-port memory or memory-mapped I/O that can be changed in ways asynchronous to the program execution. Memory sensitive mode has the following effects:

- Memory in areas where executable code resides is read from the target. When not in memory sensitive mode, MULTI assumes that executable code does not change and uses the contents of the executable file when displaying disassembly at such addresses.
- MULTI avoids reading more memory than necessary by only reading the memory locations requested and by using the specified access size. When not in memory sensitive mode, MULTI may cache some of the target memory by reading 64-byte blocks, even if only a small part of memory is actually needed.
- Only one target instruction in the Debugger pane is read and displayed at the current program location. If you want to expand the allowed range in which the Debugger can display target machine instructions, set the system variable `$_VOLATILEDISPMAX` to the desired maximum number of instruction bytes to display.

The MULTI commands **memread** and **memwrite** can also be used to perform precise memory references, even when not in memory sensitive mode. For information about these commands, see “General Memory Commands” in Chapter 11, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.

No Stack Trace Mode

To enable no stack trace mode in the Debugger, select **Debug** → **Debug Settings** → **No Stack Trace**.

No stack trace mode disables call stack viewing and related functionality. You can also select this mode by setting the system variable `$_NOSTACKTRACE=1`.

Call stacks and local variables on a call stack are not displayed because generating a call stack could cause unexpected memory references in the case where the stack pointer is either uninitialized or points to a nonstandard stack frame. As a consequence of this, certain debugging features are also disabled. You can perform instruction single-stepping, set and hit breakpoints, and start the target executing. Other debugging functionality, such as stepping over function calls, returning from function calls, and source-level single-stepping, is not supported in this mode. Once the target's stack is correctly set up, you can enable stack traces and the related Debugger features.

No stack trace mode is typically used in combination with memory sensitive mode (“Memory Sensitive Mode” on page 97) although it can be selected independently.

Executing and Controlling Your Program from the Debugger

Chapter 6

This Chapter Contains:

- Starting and Stopping a Program
- Single-Stepping Through a Program
- Using Breakpoints and Tracepoints
- Software and Hardware Breakpoint Editors
- The Breakpoints Window

This chapter explains how to use the MULTI Debugger to run programs and control and monitor processes on embedded and simulated targets. You can examine program details at different points during execution by halting the process manually, single-stepping through the program, and setting breakpoints.



Note This chapter focuses primarily on how to use MULTI Debugger features through the GUI interface. However, you can also perform most of the procedures described by issuing Debugger commands in the command pane of the Debugger window. In some situations, using Debugger commands provides more specific control or more options. For a full description of the Debugger commands, which are grouped by function, see the *MULTI: Debugging Command Reference* book.

Starting and Stopping a Program

Before you can run a program, you must be connected to an embedded target or a simulator, your executable must be associated with a connection, and your target must be prepared. If you are not connected to a target or simulator and you try to run a program, the **Connection Chooser** appears and prompts you to connect. For more information, see Chapter 4, “Connecting to Your Target”. In many cases, your executable is automatically associated with a connection and your target automatically prepared after you connect. If these things are not automatically done, MULTI prompts you to do them. For more information, see Chapter 5, “Preparing Your Target”.

The simplest way to run a program is to click  on the MULTI Debugger toolbar. The process executes and runs until it terminates successfully, encounters a serious error, is halted manually, or hits a breakpoint.

To halt a process manually, click . For instructions about using breakpoints to stop a process, see “Using Breakpoints and Tracepoints” on page 102. Click  to restart a halted process from the location where it stopped.

For an overview of the information you can view about your program and target during execution or while the process is halted, see “Viewing Program and Target Information” on page 144.



Note You can also control execution of a program—sometimes more precisely—using Debugger commands. For more information, see Chapter 14, “Program Execution Command Reference” in the *MULTI: Debugging Command Reference* book.

Single-Stepping Through a Program

Before you can single-step through a program, you must meet the same prerequisites that are listed in “Starting and Stopping a Program” on page 100.

The  and  buttons allow you to step through your program, executing one statement at a time. Both buttons execute the next single statement. The  button steps into function calls. It traces the execution of every individual instruction, even when functions are called. Conversely, the  button steps over function calls. It does not trace into called functions. If the program counter halts within a called function, click  to step out of the called function.

You can also single-step through your program using Debugger commands. For more information, see “Single-Stepping Commands” in Chapter 14, “Program Execution Command Reference” in the *MULTI: Debugging Command Reference* book.



Note When the Debugger stops on a conditionally not-executed instruction, the Debugger dims both the instruction and the PC pointer. (The PC pointer is the arrow marked with the word **STOPPED**.) In source display mode, a source instruction is dimmed if the current assembly instruction that the Debugger is stopped on, as well as any remaining assembly instructions underneath the source instruction, are conditionally not executed. In assembly display mode, the Debugger dims the instruction it is currently stopped on if the instruction is conditionally not executed. Conditionally not-executed instructions are only available on certain architectures, such as ARM, C6000, and ARC.



Note The Debugger generally completes a source-line single-step when it reaches the first instruction of a source line. The code for certain kinds of loops, such as `do { ... } while` loops and infinite loops, is generated with a backwards branch to the beginning of the loop. If debug information indicates that such a loop appears by itself on a single source line, the Debugger stops stepping after traversing the branch—even though the program may not have finished executing the entire loop—because the first instruction of the loop is

also the first instruction of the source line. This is true even if the loop appears by itself on a line because of compiler or linker code transformations such as macro expansion, inlining, or code factoring. To advance past such loops with a single command, consider using the **c** or **cb** commands with an address expression. For information about these commands, see “Continue Commands” in Chapter 14, “Program Execution Command Reference” in the *MULTI: Debugging Command Reference* book.

Using Breakpoints and Tracepoints

Breakpoints halt a process and allow you to examine your code and the state of your target at various points during program execution. You can specify conditions that trigger the breakpoint, and you can specify one or more commands to run after the breakpoint halts the process. There are two main types of breakpoints:

- *Software breakpoints* — Can be set on any executable line of code in RAM. Your process halts when it hits the software breakpoint. For more information, see “Working with Software Breakpoints” on page 105.
- *Hardware breakpoints* — Can be set in RAM, ROM, or on executable lines of code located in data memory locations. When set in data memory locations, the hardware breakpoint halts your process if it reads or writes data to the specified location. Hardware breakpoint support varies by target type. For more information, see “Working with Hardware Breakpoints” on page 110.

Breakpoints can be either temporary or permanent. MULTI deletes temporary breakpoints after your process hits them. MULTI does not delete permanent breakpoints.

You can easily set both software and hardware breakpoints by clicking the breakdots (•) that appear in the Debugger source pane. For more information, see “Breakdots, Breakpoint Markers, and Tracepoint Markers” on page 103.

On targets that use shared objects, MULTI supports breakpoints that are set in shared object code. For more information, see “Working with Shared Object Breakpoints” on page 113.

On some targets, you can also use tracepoints to collect debugging data without halting your process.



Note The MULTI Debugger also supports Debugger Notes, which allow you to attach notes to any line of code. Unlike breakpoints, Debugger Notes are not conditional and do not halt your process or execute commands. For more information, see Chapter 8, “Using Debugger Notes”.

Breakdots, Breakpoint Markers, and Tracepoint Markers

A breakdot (•) is a small dot located directly to the left of any line of source code that corresponds to an executable instruction. A breakdot indicates that you can set a software breakpoint or, on targets that support them, a hardware breakpoint or tracepoint on the line marked with the breakdot.

By default, most breakdots in the source pane are green. But you may also encounter blue or red breakdots in the following situations:

- *Blue breakdots* (•) — Indicate source lines for which the compiler generated reordered code. For example, if you step through a function with blue breakdots at the top, you will see that the program counter starts at the first green breakdot and then goes backwards to the blue breakdots before continuing on to the rest of the green breakdots. This can happen when the compiler delays the initialization of variables for a function.
- *Red breakdots* (•) — Indicate source lines that the linker removed for link-time optimization. For example, the linker may use subroutine calls and tail merges to remove redundant segments of code from object files. Because this code has been removed, you cannot set breakpoints on these source lines.



Note Link-time optimization is not available for all target processors. For more information, see the *MULTI: Building Applications* book for your target processor family.

To set a software breakpoint on any source or assembly line marked by a breakdot, click the breakdot. To set a hardware breakpoint or a tracepoint on targets that support them, right-click any breakdot on a source or assembly line and select the appropriate action from the shortcut menu that appears.

When you set a breakpoint or tracepoint, you replace the breakdot with a breakpoint or tracepoint marker. The following table describes these markers.

Breakpoint Marker	Type of Breakpoint
	Software breakpoint without conditions
	Software breakpoint with conditions
	Software breakpoint with bell activated
	Software breakpoint with command list
	Software breakpoint with breakpoint count greater than 1
	Jump breakpoint
	Any-task breakpoint (for information about any-task breakpoints in run mode, see “Any-Task Breakpoints” on page 430; for information about any-task breakpoints in freeze mode, see “Working with Freeze-Mode Breakpoints” on page 462)
	Task-specific breakpoint in freeze mode (for more information, see “Working with Freeze-Mode Breakpoints” on page 462)
	Group breakpoint for synchronized actions (for more information, see “Group Breakpoints” on page 431)
	Hardware breakpoint
	Tracepoint

If you set multiple software breakpoints on a source line, MULTI displays the marker for the enabled breakpoint that was most recently set or modified. If none of the breakpoints on the line are enabled, MULTI displays the marker for the disabled breakpoint that was most recently set.

Positioning your mouse pointer over a breakpoint or tracepoint marker displays the properties of the breakpoint or tracepoint. Clicking an active breakpoint or tracepoint marker clears the breakpoint or tracepoint and changes the marker back to a breakdot. Clicking an inactive breakpoint or tracepoint marker enables the breakpoint or tracepoint. For information about clearing and enabling multiple breakpoints and tracepoints set on a single source line, see “Multiple Breakpoints and Tracepoints on a Single Line” on page 114.

Inactive breakpoints and tracepoints have gray markers. To disable a breakpoint or tracepoint, right-click the breakpoint or tracepoint marker and select **Disable Breakpoint**, **Disable Hardware Breakpoint**, or **Disable Tracepoint**. To re-enable a disabled breakpoint or tracepoint, click the gray marker.

Middle-clicking a marker toggles the breakpoint or tracepoint between active and inactive status.

For more information about software breakpoints, see “Working with Software Breakpoints” on page 105. For more information about hardware breakpoints, see “Working with Hardware Breakpoints” on page 110.

Viewing Breakpoint and Tracepoint Information

The MULTI Debugger provides several ways for you to view information about the breakpoints and tracepoints you have set.

- **Tooltips** — In the source pane, positioning your mouse pointer over a breakpoint or tracepoint displays the properties of that breakpoint or tracepoint.
- **Breakpoints window** — The **Breakpoints** window displays and allows you to configure software breakpoints and, for targets that support them, hardware breakpoints, tracepoints, and shared object breakpoints. To open this window, do one of the following:
 - Click the **Breakpoints** button (.
 - In the Debugger command pane, enter the **bpview** command. For information about the **bpview** command, see Chapter 4, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.
 - Select **View → Breakpoints**.

Software breakpoints, hardware breakpoints, and tracepoints are displayed on separate tabs of the **Breakpoints** window.

- **Breakpoint list commands** — You can enter breakpoint commands such as **B** to print information about specific breakpoints, or all breakpoints, to the command pane. For more information about the **B** command, see Chapter 4, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

Working with Software Breakpoints

Software breakpoints halt your process when it reaches a specified line of code and meets the conditions you have specified. When the process is stopped, you

can inspect the state of your program and track down problems. For an overview of the types of information you can view about your program and target, see “Viewing Program and Target Information” on page 144. You can also specify that MULTI runs a list of commands when your process hits a breakpoint.



Note You cannot set software breakpoints in your target’s ROM because software breakpoint instructions cannot be inserted into read-only memory. However, some targets support hardware breakpoints, which can be used for debugging ROM. For more information, see “Working with Hardware Breakpoints” on page 110.

MULTI denotes a software breakpoint in the source pane with a small red stop sign (STOP). To set a software breakpoint, click a breakdot; the breakpoint marker replaces the breakdot. To move a software breakpoint, right-click the breakpoint marker and select **Move Breakpoint** (for more information, see “Moving Software Breakpoints” on page 109). To remove a breakpoint, click it; the breakpoint marker reverts to a breakdot.



Note You can also set, modify, and delete breakpoints using breakpoint commands. For a full description of these commands, see Chapter 4, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

If you set a breakpoint by clicking a breakdot, the breakpoint halts your process every time the process reaches the line of code marked with the breakpoint. After you have created a breakpoint, you can use the command pane or the **Software Breakpoint Editor** to set parameters that associate conditions and commands with the breakpoint. If you set any of these additional features, the breakpoint’s stop sign icon contains a symbol within it. Even if a breakpoint has more than one property, only one of these icons is displayed. To view these unique breakpoint icons, see the table in “Breakdots, Breakpoint Markers, and Tracepoint Markers” on page 103. For more information about the software breakpoint parameters you can set, see “Creating and Editing Software Breakpoints” on page 106.

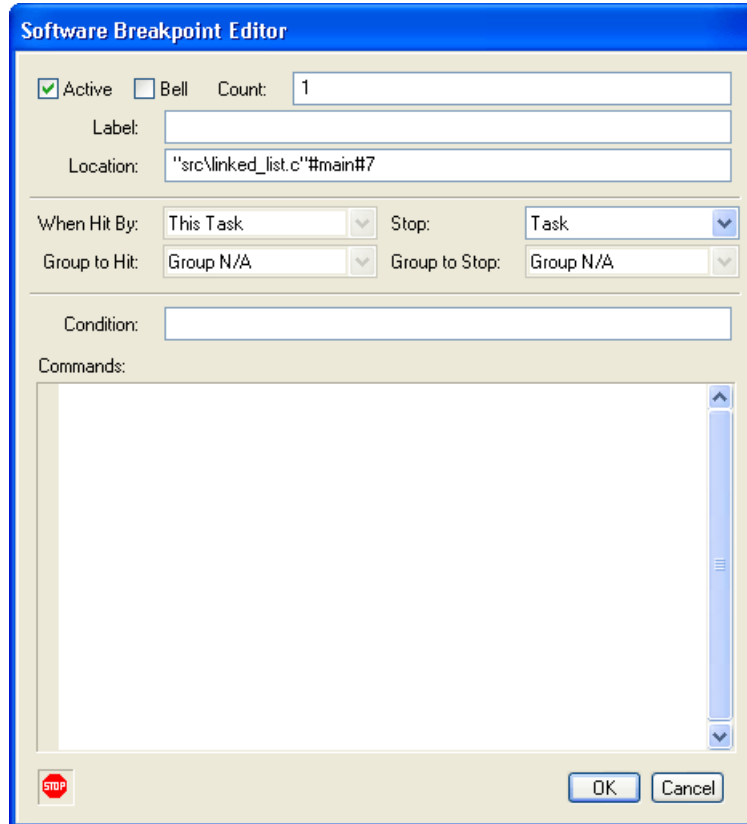
Creating and Editing Software Breakpoints

You can set a basic software breakpoint quickly and easily:

- In the source pane, click a breakdot (■). For information about breakdots, see “Breakdots, Breakpoint Markers, and Tracepoint Markers” on page 103.
- In the Debugger command pane, enter the **b** command. If you do not specify a line number, MULTI sets the breakpoint at the location of the current line pointer. For more information about this command, see Chapter 4, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

After you have created a software breakpoint, you can add conditions and command lists to it by using the **Software Breakpoint Editor**. To open a **Software Breakpoint Editor**, do one of the following.

- In the Debugger source pane, right-click the breakpoint’s stop sign icon and choose **Edit Breakpoint** from the shortcut menu.
- Click the **Breakpoints** button () to open the **Breakpoints** window, select the **Software** tab, and double-click the breakpoint you want to edit. For alternative ways to open the **Breakpoints** window, see “Viewing Breakpoint and Tracepoint Information” on page 105.
- In the Debugger command pane, enter the **editswbp** command. For more information about this command, see Chapter 4, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.



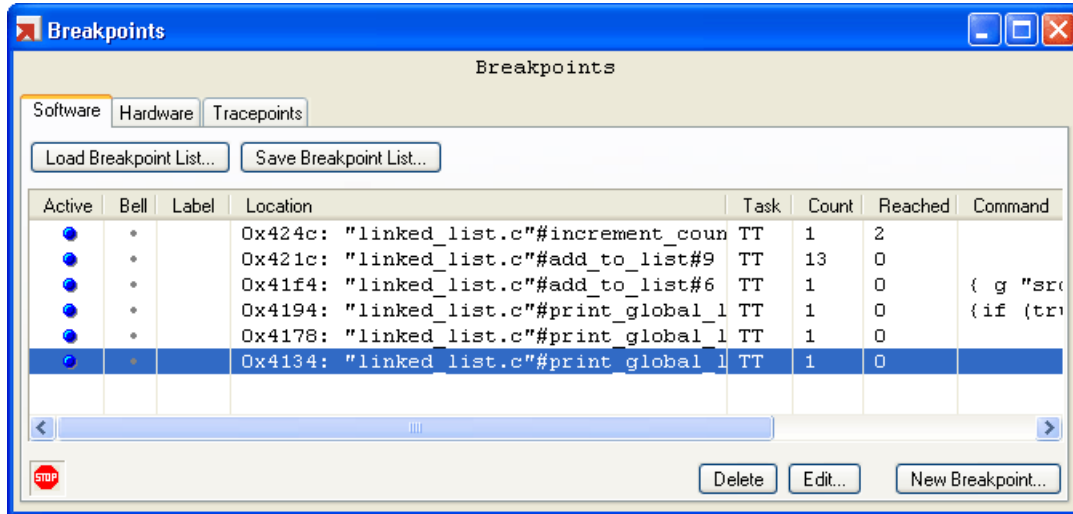
You can set and modify all the basic properties of a software breakpoint by using the fields in this window, which are described in “Software and Hardware Breakpoint Editors” on page 115.



Note You can also set and modify software breakpoint properties by entering breakpoint commands in the Debugger command pane. For detailed information about these commands, see Chapter 4, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

Viewing and Managing Software Breakpoints

To view all software breakpoints for a program and access them for editing, use the **Software** tab of the **Breakpoints** window. To open the **Breakpoints** window, click the **Breakpoints** button () located on the toolbar. (For alternative ways to open the **Breakpoints** window, see “Viewing Breakpoint and Tracepoint Information” on page 105.) Click the **Software** tab (this is the default).



The **Software** tab contains a list of your program's software breakpoints and their properties. The information displayed for each breakpoint corresponds to the breakpoint properties displayed in the **Software Breakpoint Editor**. For a description of the columns and buttons that appear in the **Breakpoints** window and the actions and shortcuts available from the window, see "The Breakpoints Window" on page 123.

Moving Software Breakpoints

You can move software breakpoints easily. To move a software breakpoint, right-click the breakpoint marker and select **Move Breakpoint**. A dialog box prompts you to either click the source line where you want to move the breakpoint or cancel the operation. MULTI signifies the source line where your mouse is currently positioned by enclosing that line's breakdot with a small square. Click one of these locations (either on the source line or on the breakdot itself) to move the breakpoint to that line. The following list provides caveats for moving breakpoints.

- You cannot move a breakpoint out of the current function in which it resides.
- In **Source Only** mode (**View** → **Display Mode** → **Source Only**), you cannot move a breakpoint from its original location if multiple breakpoints occur on the original source line. You must change the display mode to an assembly view—either **View** → **Display Mode** → **Interlaced Assembly** or **View** → **Display Mode** → **Assembly Only**. Then move the individual breakpoint using the same method as outlined previously.

- You cannot move a breakpoint to a source line that already contains a breakpoint.

Working with Hardware Breakpoints

On some targets, MULTI can set a small number of hardware breakpoints in your program. MULTI denotes a hardware breakpoint in the source pane with a small purple box (■). As with software breakpoints, you can set hardware breakpoints on your program's source or assembly lines. However, unlike software breakpoints, hardware breakpoints can be set in ROM because MULTI does not need to modify target memory to set them. You can also set hardware breakpoints on data memory locations, so that your process stops when it reads from or writes to those memory locations.

The number of hardware breakpoints you may set varies from target to target, but it is usually fewer than four. Additionally, even targets that support hardware breakpoints may not support all hardware breakpoint capabilities. For information about how your target handles hardware breakpoints if you are using a Green Hills Probe or SuperTrace Probe, see "Target-Specific Hardware Breakpoint Support" in Chapter 6, "Utilities and Troubleshooting" in the *Green Hills Debug Probes User's Guide*.

Creating and Editing Hardware Breakpoints

If you are connected to a target that supports hardware breakpoints, you can set a hardware breakpoint in any of the following ways.

- In the Debugger source pane, right-click a green breakdot (■) and choose **Set Hardware Breakpoint**.
- In the Debugger source pane, right-click a global or static variable and choose **Set Hardware Breakpoint**. (If the variable does not have a pre-defined value, you must start the program before you can set the hardware breakpoint.)
- In the Debugger command pane, enter the **hardbrk** command. For more information about this command, see Chapter 4, "Breakpoint Command Reference" in the *MULTI: Debugging Command Reference* book.
- On the **Hardware** tab of the **Breakpoints** window, click the **New HW Breakpoint** button.



Note If you frequently set hardware breakpoints in code, you may want to create a custom mouse binding so that you can set a hardware breakpoint by shift-clicking a breakdot in the source pane. For more information, see the **sethbp** command in Chapter 4, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

After you have created a hardware breakpoint, you can add conditions and command lists to it using the **Hardware Breakpoint Editor**. To open a **Hardware Breakpoint Editor**, do one of the following:

- In the Debugger source pane, right-click the breakpoint’s cube icon in the source pane and choose **Edit Hardware Breakpoint** from the shortcut menu.
- In the Debugger source pane, right-click a global or static variable and choose **Edit Hardware Breakpoint**.
- Click the **Breakpoints** button (), select the **Hardware** tab, and double-click the breakpoint you want to edit.
- In the Debugger command pane, enter the **edithwbp** command. For information about the **edithwbp** command, see Chapter 4, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

The screenshot shows the **Hardware Breakpoint Editor** dialog box. It contains the following fields and options:

- Symbol:** "linked_list.c" #main#7
- Address:** 0x42f4
- Length:** 4
- Mask:** (empty)
- Access Size:** Any (dropdown menu)
- Value:** (empty)
- Value Mask:** (empty)
- Break On:** ☐ Read ☐ Write ☒ Execute
- Type:** ☐ Global ☐ Virtual Memory
- Active:** ☒ **Rolling:** ☐ **Count:** 1
- Condition:** (empty text box)
- Commands:** (empty list box with up/down arrows)
- Buttons:** OK, Cancel

You can set and modify all the basic properties of a hardware breakpoint by using the fields in this window, which are described in “Software and Hardware Breakpoint Editors” on page 115.

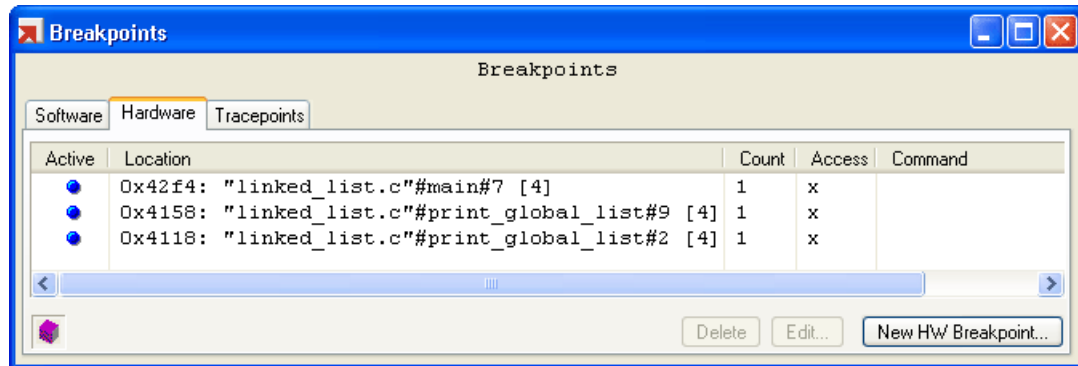


Note You can also set and modify hardware breakpoint properties by entering breakpoint commands in the Debugger command pane. For detailed information about these commands, see Chapter 4, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

Viewing and Managing Hardware Breakpoints

To work with hardware breakpoints, you must be connected to a debugging target that supports them. See the **connect** command in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.

To view all hardware breakpoints and access them for editing, use the **Hardware** tab of the **Breakpoints** window. To open the **Breakpoints** window, click the **Breakpoints** button () located on the toolbar. (For alternative ways to open the **Breakpoints** window, see “Viewing Breakpoint and Tracepoint Information” on page 105.) Click the **Hardware** tab.



The **Hardware** tab contains a list of all the hardware breakpoints in your program. For a description of the columns and buttons that appear in the **Breakpoints** window and the actions and shortcuts available from the window, see “The Breakpoints Window” on page 123.

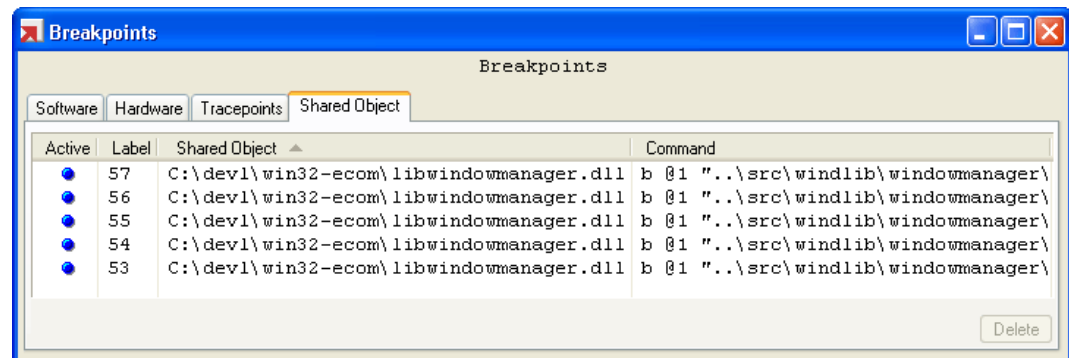
Working with Shared Object Breakpoints

On targets that use shared objects, MULTI supports breakpoints set in shared object code. Because shared object breakpoints are by definition located in object code that is not loaded, it is not possible to directly set shared object breakpoints. MULTI provides only the ability to list and delete these breakpoints and toggle them on and off.

Viewing and Managing Shared Object Breakpoints

When a shared object containing breakpoints is removed from the target process at run time, the breakpoints are shown in the **Shared Object** tab of the **Breakpoints** window. To open the **Breakpoints** window, click the **Breakpoints** button () located on the toolbar. (For alternative ways to open the **Breakpoints** window, see “Viewing Breakpoint and Tracepoint Information” on page 105.)

The **Shared Object** tab is only visible when shared object breakpoints are available for display.



The list of shared object breakpoints specifies whether each breakpoint is active or inactive, the unique label of each breakpoint, the shared object that each breakpoint is set in, and the command that restores each breakpoint when the shared object is loaded again. For a comprehensive description of the **Breakpoints** window and the actions and shortcuts available from the window, see “The Breakpoints Window” on page 123.



Note To use the Debugger’s command pane to list shared object breakpoints, enter the **B** command. See the **B** command in Chapter 4, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

Advanced Breakpoint Topics

Multiple Breakpoints and Tracepoints on a Single Line

If some combination of software breakpoints, hardware breakpoints, and tracepoints are all set on the same source line and you click the marker to the left of the source line, *all* software breakpoints are enabled or cleared independently of hardware breakpoints and tracepoints. Similarly, *all* hardware breakpoints are enabled or cleared independently of software breakpoints and tracepoints. The same principle applies to tracepoints. Breakpoints and tracepoints are enabled or cleared based on the following criteria:

- If all software breakpoints, hardware breakpoints, or tracepoints are disabled, MULTI enables all software breakpoints, hardware breakpoints, or tracepoints, respectively.
- If at least one software breakpoint, hardware breakpoint, or tracepoint is enabled, MULTI clears all software breakpoints, hardware breakpoints, or tracepoints, respectively.

For example, suppose a single source line contains multiple software breakpoints, some of which are enabled and some disabled; one hardware breakpoint that is enabled; and one tracepoint that is disabled. If you click the marker to the left of the source line, MULTI clears all the software breakpoints, clears the hardware breakpoint, and enables the tracepoint.

Breakpoint Limitations

- Full source-level debugging is not possible within procedure prologues and epilogues, so MULTI does not display source-level breakpoints within these regions. For more information, see “The Call Stack Window and Procedure Prologues and Epilogues” on page 377.
- If you halt a process at an instruction where a software breakpoint is set, the software breakpoint may not be hit. This means that any operation associated with the breakpoint is not executed. For example, any commands set to run when the breakpoint is hit will not run; if the breakpoint is a jump breakpoint, it will not jump to the specified location; etc.

Software and Hardware Breakpoint Editors

The following table describes the fields and options available in the **Software Breakpoint Editor** and the **Hardware Breakpoint Editor**. The options are ordered alphabetically in the table below. Unless otherwise stated, all descriptions of toggle options explain the behavior of the option when enabled.

Editor Field	Effect
Access Size	Specifies the access size to break on. This option is available in the Hardware Breakpoint Editor .
Active	Activates the breakpoint. To inactivate the breakpoint, clear this box. By default, breakpoints are active. When a breakpoint is active, the process stops when it hits the breakpoint and it meets the conditions of the breakpoint. When a breakpoint is inactive, it has no effect on the process. Unlike deleting a breakpoint, inactivating a breakpoint disables the breakpoint temporarily; it does not delete the breakpoint and its parameters. See also the tog command in Chapter 4, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. This option is available in the Software Breakpoint Editor and the Hardware Breakpoint Editor .
Address	Specifies the location of the hardware breakpoint as an address. To set the location of your breakpoint, enter an address, such as 0x001000. When your process accesses memory at this address, it hits the hardware breakpoint and stops. Note that the Length , Mask , Access Size , Value , and Value Mask fields affect which memory accesses cause your process to hit the hardware breakpoint. If the address you enter is that of a valid symbol in your program, MULTI automatically enters the symbol name in the Symbol field. This option is available in the Hardware Breakpoint Editor .

Editor Field	Effect
Bell	Enables a bell that beeps each time the process hits the software breakpoint. By default, the bell is disabled. To enable the bell, check this box. If the breakpoint does not stop the process, MULTI does not beep when it reaches the breakpoint, even if the breakpoint's bell is activated. Situations when this could occur include the following: • The breakpoint is disabled. • The breakpoint is conditional, and its condition evaluated to false. • The breakpoint is associated with a command that continues execution of the program. This option is available in the Software Breakpoint Editor.
Break On	Specifies when your process stops. Click the memory access type that represents when your process should stop. The access types are: • Read — Your process stops whenever it reads from memory at the hardware breakpoint location. • Write — Your process stops whenever it writes to memory at the hardware breakpoint location. • Execute — Your process stops whenever it fetches instructions from memory at the hardware breakpoint location. You should not set Read and/or Write in conjunction with Execute. This option is available in the Hardware Breakpoint Editor.
Commands	Specifies one or more commands to run every time your process hits the breakpoint. You can enter multiple commands by inserting a semicolon between commands or by entering each command on a separate line in the Commands text box. This option is available in the Software Breakpoint Editor and the Hardware Breakpoint Editor.

Editor Field	Effect
Condition	Specifies the condition(s) under which the breakpoint halts your process. To set conditions for a breakpoint, enter a language expression in this field. While the condition remains false (or zero), the breakpoint does not stop your process. If your process hits the breakpoint when the condition is true (or not zero), the process stops. For software breakpoints, the condition is evaluated in the context where the breakpoint appears. For hardware breakpoints, MULTI evaluates the condition in a global context, so you can only use global variables here. Even when the condition is false, the breakpoint briefly interrupts the process so that MULTI can evaluate the condition. Therefore, a code segment containing a conditional breakpoint executes more slowly than one without such a breakpoint, even when the condition is false. This option is available in the Software Breakpoint Editor and the Hardware Breakpoint Editor.
Count	Specifies the breakpoint's count, which is the number of times the process must hit the breakpoint before MULTI halts the process. The default breakpoint count is 1. If the breakpoint's count is 1, the process stops when it hits the breakpoint. If the count is greater than 1, the process does not stop until it has hit the breakpoint the number of times specified in this field, after which the process halts every subsequent time it hits the breakpoint. Each time the process hits the breakpoint, the breakpoint count is decremented until it reaches 1. For more information about breakpoint counts, see Chapter 4, "Breakpoint Command Reference" in the <i>MULTI: Debugging Command Reference</i> book. This option is available in the Software Breakpoint Editor and the Hardware Breakpoint Editor.
Group to Hit	Identifies what task group must hit the software breakpoint for the breakpoint to halt the process. This option is only available if your target supports task groups and you set the field When Hit By to <code>Any Task in Group</code>. This option is available in the Software Breakpoint Editor.

Editor Field	Effect
Group to Stop	Specifies what task group halts when a process hits the software breakpoint. The Group to Stop option is only available if you set the Stop field to Task Group on a target that supports task groups. The Group to Stop option is available in the Software Breakpoint Editor.
Label	Specifies a string you can use to refer to the software breakpoint later. Breakpoint labels are optional, but can be useful when you enter breakpoint commands in the command pane. Examples of breakpoint commands include b, d, and tog. For information about these and other breakpoint commands, see Chapter 4, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. To assign a breakpoint label, type a word or name in the text field. Breakpoint labels cannot contain spaces or special characters. For more information about using breakpoint labels with commands, see Chapter 2, “Using Debugger Commands” in the <i>MULTI: Debugging Command Reference</i> book. This option is available in the Software Breakpoint Editor.
Length	Specifies the size of memory your process accesses at the specified breakpoint address. Your process hits a hardware breakpoint only if it accesses Length number of bytes at the specified address. For example, if Length is 2 bytes, then your process hits the breakpoint when a short value is written to the breakpoint address, but does not hit the breakpoint when a char or int value is written to the same address. This option is available in the Hardware Breakpoint Editor.
Location	Specifies the location of a software breakpoint in your program’s code. This is a required property of all software breakpoints, so this field must contain a valid address expression. For more information, see “Using Address Expressions in Debugger Commands” in Chapter 2, “Using Debugger Commands” in the <i>MULTI: Debugging Command Reference</i> book. This option is available in the Software Breakpoint Editor.

Editor Field	Effect
Mask	Specifies the address bits that MULTI uses when comparing the current address to the hardware breakpoint address. To compare addresses, MULTI does the following: 1. Performs a bitwise AND operation on the mask and the hardware breakpoint address. 2. Performs a bitwise AND operation on the mask and the current address. 3. Compares the results of these two operations. If the results are equal and the Value and Value Mask fields are not set, MULTI triggers the hardware breakpoint. If the Value and Value Mask fields are set, MULTI verifies them before triggering the hardware breakpoint. For example, suppose: • You define mask as: 0xFF0 (1111 1111 0000) • You define the hardware breakpoint address as: 0xCD6 (1100 1101 0110) • You do not define the Value or Value Mask Because the bitwise AND operations cause the last 4 bits of the current address and the breakpoint address to be 0000, MULTI triggers the hardware breakpoint when your process accesses memory between addresses 0xCD0 (1100 1101 0000) and 0xCDF (1100 1101 1111). The default mask is 0xFFFFFFFF, which specifies that the current address must be identical to the specified breakpoint address. This option is available in the Hardware Breakpoint Editor.
Rolling	Allows MULTI to delete the breakpoint if MULTI needs the breakpoint's resources. This option is not selected by default. This option is available in the Hardware Breakpoint Editor.

Editor Field	Effect
Stop	Specifies where a process stops when it hits the software breakpoint. The available settings for this option are: • Task — The breakpoint stops the task that hits that breakpoint. • System — The breakpoint stops the whole system. • Task Group — The breakpoint stops all the tasks in the group specified by the Group to Stop field. If your target does not support these breakpoints, this setting is disabled and cannot be modified. This option is available in the Software Breakpoint Editor.
Symbol	Specifies the location of the hardware breakpoint with a symbol. When your process accesses the symbol, the process hits the hardware breakpoint and stops. To enter a symbol to serve as the location of your hardware breakpoint, type the name of a variable or other valid symbol into this field. MULTI evaluates the symbol in a global context, so it is typical to use global variables here. When you enter a valid symbol, MULTI automatically enters the address and size of the symbol in the Address and Length fields. If you want to specify an address manually instead of using a symbol to specify the location of the hardware breakpoint, use the Address field. This option is available in the Hardware Breakpoint Editor.
Type	Specifies a type for the hardware breakpoint. The available settings for this option are: • Global — Specifies that the hardware breakpoint can be hit by any task in the address space in which it is set. This setting is only applicable if you are using a run-mode connection to debug INTEGRITY (version 10 or later) or VxWorks. • Virtual Memory — Instructs INTEGRITY to use a virtual memory breakpoint to simulate the hardware breakpoint. This setting is only applicable if you are using a run-mode connection to debug INTEGRITY (version 10 or later). For more information, see the hardbrk command's global and vm options in Chapter 4, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. This option is available in the Hardware Breakpoint Editor.

Editor Field	Effect
Value	Compares the specified value with the value located at the hardware breakpoint address, when the process has accessed memory at that address (see the preceding Address description). If the specified value, when masked with Value Mask is equal to the value located at the hardware breakpoint address, the hardware breakpoint triggers. (See the following Value Mask description.) MULTI verifies Value and Value Mask after verifying Address and Mask. This setting is only applicable on certain targets and debug servers. If you set this field and it is not applicable in your current environment, MULTI either returns an error or the system ignores the setting. When used for reads and writes less than 32 bits in size, the behavior of this field is target dependent. The behavior may also vary between big and little endian targets. This option is available in the Hardware Breakpoint Editor.

Editor Field	Effect
Value Mask	Specifies the value bits that MULTI uses when comparing the current value to the hardware breakpoint value. To compare values, MULTI does the following: 1. Performs a bitwise AND operation on the value mask and the hardware breakpoint value. 2. Performs a bitwise AND operation on the value mask and the current value. 3. Compares the results of these two operations. If the results are equal, MULTI triggers the hardware breakpoint. The default value mask is <code>0xFFFFFFFF</code>, which specifies that the current value must be identical to the specified breakpoint value. MULTI verifies Value and Value Mask after verifying Address and Mask. This setting is only applicable on certain targets and debug servers. If you set this field and it is not applicable in your current environment, MULTI either returns an error or the system ignores the setting. When used for reads and writes less than 32 bits in size, the behavior of this field is target dependent. The behavior may also vary between big and little endian targets. This option is available in the Hardware Breakpoint Editor.
When Hit By	Identifies what must hit the software breakpoint for the breakpoint to halt the process. You can specify that the process stops when one of the following hits the breakpoint: • This Task — A specific task • Unattached Tasks — Any unattached task • Attached Tasks — Any attached task • Any Task — Any task • Any Task in Group — Any task in the group specified in the Group to Hit field If your target does not support these breakpoint conditions, this setting is disabled and cannot be modified. This option is available in the Software Breakpoint Editor.

The Breakpoints Window

The following sections describe the columns, buttons, and shortcut menus available in the **Breakpoints** window, as well as the actions you can perform from this window. Only the **Software**, **Hardware**, and **Shared Object** tabs are covered here.



Note Most of the actions you can perform from the **Breakpoints** window, you can also perform by entering commands in the Debugger command pane. For more information, see Chapter 4, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

Breakpoints Window Columns

The following table describes breakpoint properties and how they are displayed in the **Breakpoints** window.

Columns vary according to the tab selected (**Software**, **Hardware**, or **Shared Object**). The columns are ordered alphabetically in the table below.

Access	The memory access type, which causes your process to trigger the hardware breakpoint. Access types are: • r — (read) Your process stops whenever it reads from memory at the hardware breakpoint location. • w — (write) Your process stops whenever it writes to memory at the hardware breakpoint location. • rw — (read/write) Your process stops whenever it either reads from or writes to memory at the hardware breakpoint location. • x — (execute) Your process stops whenever it fetches instructions from memory at the hardware breakpoint location. This column is available on the Hardware tab.
Active	A large blue dot (●) indicates the breakpoint is active. A small gray dot (•) indicates the breakpoint is inactive. To toggle between the two states, click the dot. This column is available on the Software, Hardware, and Shared Object tabs.

Bell	A large blue dot (●) indicates the bell is active. A small gray dot (•) indicates the bell is inactive. To toggle between the two states, click the dot. This column is available on the Software tab.
Command	On the Software and Hardware tabs: the commands that run every time your process hits the breakpoint. On the Shared Object tab: the command that restores each shared object breakpoint when the shared object is loaded again.
Count	If MULTI is counting the number of times the process hits the breakpoint (this is the common case): the breakpoint's current count, which indicates how many more times the process must hit the breakpoint before the process halts. If the target is counting the number of times the process hits the breakpoint (for example, if you are using INTEGRITY 10 with a run-mode connection): the behavior of this column is undefined. This column is available on the Software and Hardware tabs.
Label	The label (if any) of the breakpoint. Breakpoint labels cannot contain spaces or special characters. This column is available on the Software and Shared Object tabs.
Location	The location of the breakpoint in your program's code. For more information, see "Using Address Expressions in Debugger Commands" in Chapter 2, "Using Debugger Commands" in the <i>MULTI: Debugging Command Reference</i> book. This column is available on the Software and Hardware tabs.
Reached	If MULTI is counting (this is the common case): the number of times your process has hit the breakpoint. If the target is counting (for example, if you are using INTEGRITY 10 with a run-mode connection): the behavior of this column is undefined. This column is available on the Software tab.
Shared Object	The shared object that each breakpoint is set in. This column is available on the Shared Object tab.
Task	Identifies what must hit the breakpoint for the breakpoint to halt your process, and also what tasks halt when your process hits the breakpoint. This column is available on the Software tab.

Breakpoints Window Buttons

In addition to displaying information about all your breakpoints, the **Breakpoints** window contains buttons that make it easy for you to create, modify, load, save, or delete breakpoints. Several of these actions can be performed on multiple breakpoints simultaneously.

Buttons vary according to the tab selected (**Software**, **Hardware**, or **Shared Object**). The buttons are ordered alphabetically in the following table.

Button	Action
Delete	Deletes the selected breakpoint(s). This button is available on the Software , Hardware , and Shared Object tabs.
Edit	Opens the Software Breakpoint Editor or the Hardware Breakpoint Editor , which you can use to edit the selected software breakpoint or hardware breakpoint, respectively. For more information about setting software breakpoint parameters, see “Creating and Editing Software Breakpoints” on page 106. For more information about setting hardware breakpoint parameters, see “Creating and Editing Hardware Breakpoints” on page 110. This button is available on the Software and Hardware tabs.
Load Breakpoint List	Opens a Load Breakpoints dialog box where you can choose to load the file in which you previously saved breakpoints. (See Save Breakpoint List below.) Note: The Debugger may not be able to load group breakpoints. This button is available on the Software tab.
New Breakpoint or New HW Breakpoint	Creates a new breakpoint at the location of the current line pointer and opens a Software Breakpoint Editor or Hardware Breakpoint Editor on the new breakpoint. This button is available on the Software and Hardware tabs.
Save Breakpoint List	Opens a Save Breakpoints dialog box where you can choose a file in which to save your program’s currently set breakpoints. You can reload the file later. Note that the Debugger may not be able to reload group breakpoints. (See Load Breakpoint List above.) This button is available on the Software tab.

Breakpoints Window Mouse and Keyboard Actions

The following table describes the mouse and keyboard actions you can perform on items located in the **Breakpoints** window. Possible actions vary according to the tab selected (**Software**, **Hardware**, or **Shared Object**). The menu options are ordered alphabetically in the following table.

Action	Result
Click a breakpoint	Displays the breakpoint's location in the Debugger, if the breakpoint is set in source code. This action is possible on the Software and Hardware tabs.
Double-click a breakpoint	Opens the Software Breakpoint Editor or the Hardware Breakpoint Editor , which you can use to edit the selected software breakpoint or hardware breakpoint, respectively. This action is possible on the Software and Hardware tabs.
Click the Active dot	Toggles the state of the selected breakpoint between active and inactive. See also the tog command in Chapter 4, "Breakpoint Command Reference" in the <i>MULTI: Debugging Command Reference</i> book. This action is possible on the Software , Hardware , and Shared Object tabs.
Click the Bell dot	Toggles the state of the bell between enabled and disabled. If enabled on a specific breakpoint, the bell beeps every time the process stops at that breakpoint. This action is possible on the Software tab.
Right-click a breakpoint	Opens a shortcut menu. Available menu options are described in the following table. This action is possible on the Software , Hardware , and Shared Object tabs.
Press Enter	Opens the Software Breakpoint Editor or the Hardware Breakpoint Editor , which you can use to edit the selected software breakpoint or hardware breakpoint, respectively. This action is possible on the Software and Hardware tabs.
Press Delete	Deletes the selected breakpoint(s). This action is possible on the Software , Hardware , and Shared Object tabs.

Breakpoints Window Shortcut Menu

The following table describes the menu options that appear when you right-click a breakpoint in the **Breakpoints** window. Menu options vary according to the type of breakpoint clicked (software, hardware, or shared object). The menu options are ordered alphabetically in the following table.

Menu Item	Action
Beep on Selected Breakpoints	Enables the breakpoint bell for the selected breakpoints. A beep sounds when the process stops at these specified breakpoints. This menu item is available from the Software tab shortcut menu.
Delete Selected Breakpoints or Delete Selected Shared Object Breakpoints	Deletes the selected breakpoint(s). This menu item is available from the Software , Hardware , and Shared Object tab shortcut menus.
Disable Selected Breakpoints or Disable Selected Shared Object Breakpoints	Inactivates all the selected breakpoints so that the process does not stop when it hits them. See also the tog command in Chapter 4, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. This menu item is available from the Software , Hardware , and Shared Object tab shortcut menus.
Do not Beep on Selected Breakpoints	Disables the breakpoint bell for the selected breakpoints. No sound is emitted when the process stops at these specified breakpoints. This menu item is available from the Software tab shortcut menu.
Edit Selected Breakpoint	Opens the Software Breakpoint Editor or the Hardware Breakpoint Editor , which you can use to edit the selected software breakpoint or hardware breakpoint, respectively. This menu item is available from the Software and Hardware tab shortcut menus.
Enable Selected Breakpoints or Enable Selected Shared Object Breakpoints	Activates all the selected breakpoints so that the process stops when it hits them. See also the tog command in Chapter 4, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. This menu item is available from the Software , Hardware , and Shared Object tab shortcut menus.

Menu Item	Action
New Software Breakpoint or New Hardware Breakpoint	Opens the Software Breakpoint Editor or the Hardware Breakpoint Editor, which you can use to create a new software or hardware breakpoint with the properties that you specify. This menu item is available from the Software and Hardware tab shortcut menus.
Show In Debugger	Displays the source code where the selected breakpoint is set. See also the e command in Chapter 12, “Navigation Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. This menu item is available from the Software and Hardware tab shortcut menus.

Navigating Windows and Viewing Information

Chapter 7

This Chapter Contains:

- Navigating and Searching Basics
- Navigating, Browsing, and Searching the Source Pane
- Viewing Program and Target Information

This chapter describes how to navigate and browse MULTI Debugger windows.

Navigating and Searching Basics

The following sections explain features and procedures common to most MULTI Debugger windows.

Using the Scroll Bar

- You can either use the scroll bar or enter the **scrollcommand** command to scroll through your program. For more information about this command, see Chapter 12, “Navigation Command Reference” in the *MULTI: Debugging Command Reference* book.
- For information about navigating the source pane, see “Navigating, Browsing, and Searching the Source Pane” on page 136.

Infinite Scrolling

In two situations, normal scrolling behavior is replaced by *infinite scrolling*. In this mode, the thumb in the vertical scroll bar stays fixed in the middle of the scroll bar, does not reflect the relative position of the display, and cannot be dragged to new locations. The scroll thumb’s appearance is also different. On Windows, the Debugger sets the thumb at its minimum size. On UNIX, the Debugger replaces the thumb with a diamond. You can still click above or below the thumb or use the arrow keys to scroll up or down. Infinite scrolling occurs in the following situations:

- When the Debugger displays only assembly code in the source pane. See “Source Pane Display Modes” on page 41.
- In a **Memory View** window. See Chapter 13, “Using the Memory View Window”.

Incremental Searching

You can perform incremental text searches in most MULTI debug windows. (In some windows, you must select something before you can search.) To start an incremental forward search, press Ctrl + F. To start an incremental backward

search, press Ctrl+B. When you start a search, `Srch` appears on the left side of the window's status bar.

After you press Ctrl+F or Ctrl+B and start typing, MULTI begins searching the active window for the string of characters and highlights the first match it finds. If subsequent keystrokes do not match the first selection, MULTI continues to search the active window for an exact match. The search string appears to the right of `Srch` in the status bar. In most windows, the search starts with the text that the window currently displays. In the source pane, the search begins at the current line pointer.

To find the next or previous search match, press Ctrl+F or Ctrl+B again. Incremental searches wrap around the entire text buffer of the window you are searching. When the Debugger reaches the end or the beginning of the window buffer, it beeps to indicate that it is about to wrap.

To end a search, press Esc or Enter.

You can switch the search mode for a single search from case-insensitive to case-sensitive by typing uppercase characters in the search string. To change the default case sensitivity for all incremental searches in the current session, enter the **chgcse** command in the Debugger command pane. For more information about this command, see Chapter 16, "Search Command Reference" in the *MULTI: Debugging Command Reference* book.

The following table summarizes the shortcuts and keystrokes available for incremental searching.

Keyboard Shortcut	Effect
Ctrl + F	Starts an incremental forward search or, if you have already started a search, advances to the next matching pattern. If you have not started searching and you press Ctrl + F twice, you resume the last search.
Ctrl + B	Starts an incremental backward search or, if you have already started a search, jumps backward to the previous matching pattern. If you have not started searching and you press Ctrl + B twice, you resume the last search.
Ctrl + U (while in search mode)	Resets the search pattern.

Keyboard Shortcut	Effect
Esc or Enter (while in search mode)	Ends the search. In source pane searches, the last match remains selected.
<i>any_character</i> (while in search mode)	Adds a character to the search pattern.
Backspace (while in search mode)	Deletes the last character in the search pattern.



Note You can also enter search commands in the command pane or use the **Source Pane Search** dialog box to perform incremental searches in the source pane. For more information, see Chapter 16, “Search Command Reference” in the *MULTI: Debugging Command Reference* book and “Using the Source Pane Search Dialog Box” on page 143.

Example 1. Searching for a String

If the source pane of an active Debugger window contains the following text:

```
this is a search string.
```

and you start a search by pressing Ctrl+F and typing the letter `i`, the Debugger highlights the character `i` in the word `this`.

If you then type `s`, the Debugger highlights the two characters `i` and `s` in the word `this`.

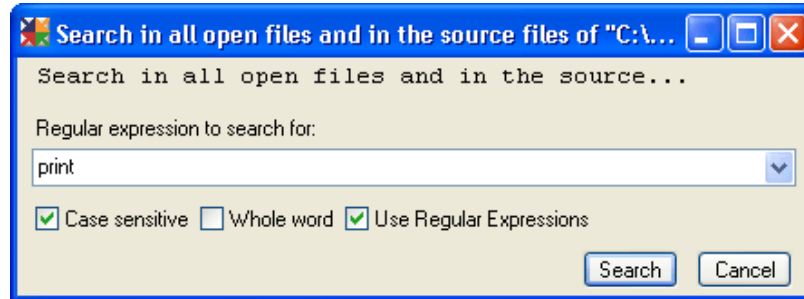
To jump to the next occurrence of the pattern `is`, press Ctrl+F again. The Debugger selects the word `is`.

To search next for the pattern `in`, press Backspace to reset the search string to `i`, and then type `n`. The Debugger highlights the characters `i` and `n` in `string`.

To stop the search, press Enter. The `in` in `string` remains selected.

Searching in Files

The MULTI Debugger supports full-text searching of open files and, if debugging information is available, of all the files that make up a program. To search your source files, select **Tools** → **Search in Files**. A Search in Files dialog box appears.



Enter your search string in the text box or use the drop-down list to select recent search strings. You can also select any of the following optional search criteria.

- **Case sensitive** — The search only finds text that matches the case of the search string exactly. If this box is cleared, the search ignores case when searching for a match. By default, this box is checked.
- **Whole word** — The search only finds text that contains the search terms as words. This means that the matching string must be preceded by a non-word character and followed by a non-word character, where word characters are letters, digits, and the underscore. For example, if you select this check box, a search for `ice` does not match `slice` or `ice__`, but it does match `ice-9`. This box is cleared by default.
- **Use Regular Expressions** — The search treats the text you enter as a regular expression. If this box is cleared, the search treats the text you enter as a fixed string. By default, this box is checked.

After you have entered the search string and set the search criteria, click **Search** to perform the search. A **Search in Files Results** window opens to show the search progress. For a description of this window, see “Viewing Search in Files Results” in Chapter 2, “The MULTI Editor” in the *MULTI: Editing Files and Configuring the IDE* book.



Note To access the same Search in Files capability without using a GUI interface, use the **grep** command. For more information about this command, see Chapter 16, “Search Command Reference” in the *MULTI: Debugging Command Reference* book.



Note The Search in Files capability runs the BSD **grep** utility. A copy of BSD **grep** is installed along with MULTI. However, BSD **grep** is not part of MULTI and is not distributed under the same license as MULTI. For more information about the license under which BSD **grep** is distributed, refer to the file **bsdgrep.txt**, which is located in the **copyright** subdirectory of the MULTI installation directory. For information about the search expression format that BSD **grep** uses, refer to the OpenBSD `re_format(7)` man page.

Selecting, Cutting, and Pasting Text

In most MULTI windows, you can select and copy text using your mouse and common keyboard shortcuts. Some windows, however, do not support text selection, copying, and/or pasting. For more information about whether a specific window supports text selection and modification, use the index to find the main discussion of that window in this book. Conventions for selecting, copying, and pasting text differ in the source and command panes of the main Debugger window. For more information, see “Selecting, Copying, and Pasting Text in the Main Debugger Window” on page 135.

The following table describes general conventions for selecting, cutting, and pasting text in MULTI Debugger windows.

To	Do this
Select text	Use your mouse pointer to highlight text.
Copy text	Select text, and press Ctrl + C.
Paste text	Copy text, click in the spot where you want to paste the text, and press Ctrl + V.
Deselect text	Click in an area of the window without any text.



Note On Windows, the copy and paste functions use the Windows clipboard, so you can copy and paste text among any applications that use the Windows clipboard.

On UNIX, you can copy and paste your selection to other X Window System applications that support pasting, such as xterms. However, different applications may behave differently, so consult the reference manual for your specific system. Generally, when you select text, MULTI automatically copies it. Middle-click to paste automatically copied text. Text cannot be selected in two windows at the same time.

Selecting Items from Lists

In those windows containing lists of items that you can select, the following conventions usually apply:

- To select an item, single-click the row containing that item. This highlights the row.
- To select multiple, non-adjacent items/rows, hold the Ctrl key while clicking each item or row.
- To select a range of adjacent rows, click the first row, hold the Shift key, and then click the last row in the range.

Selecting, Copying, and Pasting Text in the Main Debugger Window

Conventions for selecting, copying, and pasting text in the source and command panes differ from the conventions used in other Debugger windows. In the source pane, your selection automatically expands to include a whole entity, such as a word or expression. For example, to select a word in the source pane, simply click anywhere within the word. To select all the text occurring between a pair of parentheses, including the parentheses themselves, drag your mouse over one of the parenthesis markers and release.

When selecting text in the source pane, you may find that the Debugger performs a command as soon as you release the mouse button. To prevent the Debugger from issuing a command, hold down the Ctrl key while selecting text. This also prevents the Debugger from attempting to auto-complete your selection. Alternatively, you can configure Debugger key bindings to prevent the Debugger from issuing commands. See “Customizing Keys and Mouse Behavior” in Chapter 8, “Configuring and Customizing MULTI” in the *MULTI: Editing Files and Configuring the IDE* book.

When you select text in the source pane, the Debugger automatically copies your selection. To paste the selection into the command pane, middle-click in the command pane. To enable this behavior on Windows, select **Config** → **Options** → **MULTI Editor** tab, and check **Allow middle click to paste**. In general, this behavior is similar to selecting and pasting procedures on UNIX.



Note On Windows, the automatic copy that the Debugger makes at the time of selection only allows you to paste to the command pane. To copy text and paste it into another window, highlight the text and press Ctrl + C to copy it. Then press Ctrl + V to paste it into another window.

Navigating, Browsing, and Searching the Source Pane

You can use the following shortcuts, buttons, commands, and menu items to navigate through your program code in the source pane. For information about scrolling, see “Using the Scroll Bar” on page 130.

To	Do (one of) the following
Scroll up by one screen	• Press PageUp.
Scroll down by one screen	• Press PageDown.
Scroll up by one line	• Press Shift + UpArrow.
Scroll down by one line	• Press Shift + DownArrow.
View the code one level higher on the call stack	• Press Ctrl + + (plus sign). • Click . • Select View → Navigation → UpStack.
View the code one level lower on the call stack	• Press Ctrl + - (minus sign). • Click . • Select View → Navigation → DownStack.

To	Do (one of) the following
View the procedure where the process is stopped	• Click . • In the command pane, enter the E command. For information about the E command, see Chapter 9, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. • Select View → Navigation → Current PC.
View the first procedure higher on the call stack that has source code (for example, if you are stopped inside a library function with no source code and you want to return to viewing your program)	• In the command pane, enter the uptosource command. For information about the uptosource command, see Chapter 12, “Navigation Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. • Select View → Navigation → UpStack To Source.

To	Do (one of) the following
Navigate to a specific file, procedure, line number, breakpoint, or other location	• In the command pane, enter the e command followed by an address expression. For information about the e command, see Chapter 12, “Navigation Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. • On the navigation bar, use the File Locator to navigate to a file. See “Using the File Locator” on page 139. • On the navigation bar, use the Procedure Locator to navigate to a procedure. See “Using the Procedure Locator” on page 140. • Select View → Navigation → Goto Location and enter an appropriate location in the dialog box.
Return to a location you recently viewed in the source pane	• In the command pane, enter the indexprev and indexnext commands. For information about these commands, see Chapter 12, “Navigation Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. • On the navigation bar, use the Back () and Forward () history buttons.

Using the File Locator

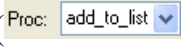
The File Locator () shows the name of the file currently displayed in the Debugger and allows you to navigate quickly to other files in your program. The following table summarizes what you can do with the File Locator.

To	Do this
View the full name and path of the displayed file	Move the cursor over the File Locator. The file path appears in a tooltip.
Navigate to another source file in your program	Type the name of the file in the File Locator and press Enter. If the name you type is of a valid source file that was compiled into your program, the Debugger navigates to the file and displays it in the source pane. If the name you type does not match a filename exactly, the Debugger displays a file whose name begins with the name that you typed, or if there is more than one match, it displays a list of files. If you do not remember the exact name of the file you are looking for, use a wildcard (see “Wildcards” on page 284). For example, if you know that a file’s name contains the word <code>sort</code>, enter <code>*sort*</code> in the File Locator to browse a list of all the files in your program that contain the word <code>sort</code> in their names. If only one file in your program matches the pattern that you type, the Debugger displays that file immediately. You can use any number of wildcard characters in your search. Wildcard searching is not case-sensitive.

To	Do this
View a file recently displayed in the source pane	Click the arrow on the right side of the File Locator to open a drop-down list of recently viewed files. The File Locator sorts the files according to when you last viewed them, with the currently displayed file at the top of the list. To display a file in the source pane, select it in the list.
Browse a list of all the files in your program	Do one of the following: • In the command pane, enter the browse files command. For information about the browse command, see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. • Click the arrow on the right side of the File Locator and select Browse all source files in program from the list that appears. • Select Browse → Files. Performing any of the above actions opens a Source Files with Procedures window. To display a file in the source pane, click the name of the file in the list.

See also “Browsing Source Files” on page 211.

Using the Procedure Locator

The Procedure Locator () shows the name of the procedure currently displayed in the Debugger and allows you to navigate quickly to other procedures in your program. The following table summarizes what you can do with the Procedure Locator.

To	Do this
View the full name and signature of the displayed procedure	Move the mouse pointer over the Procedure Locator. The full name and signature appears in a tooltip. Full signature display is only available for C++ procedures. The return type of the procedure is not included in the signature display.

To	Do this
Navigate to another procedure in your program	Type the name of the procedure in the Procedure Locator and press Enter. If the name you type is of a valid procedure that was compiled into your program, the Debugger navigates to the procedure and displays it in the source pane. If the name you type does not match a procedure name exactly, the Debugger displays a procedure whose name begins with the name that you typed, or if there is more than one match, it displays a list of procedures. If you do not remember the exact name of the procedure you are looking for, use a wildcard (see “Wildcards” on page 284). For example, if you know that a procedure’s name contains the word <code>sort</code>, enter <code>*sort*</code> in the Procedure Locator to browse a list of all the procedures in your program that contain the word <code>sort</code> in their names. If only one procedure in your program matches the pattern that you type, the Debugger displays that procedure immediately. You can use any number of wildcard characters in your search. Wildcard searching is not case-sensitive.
View a procedure recently displayed in the source pane	Click the arrow on the right side of the Procedure Locator to open a drop-down list of recently viewed files. The Procedure Locator sorts the procedures according to when you last viewed them, with the currently displayed procedure at the top of the list. If more than one procedure is visible in the source pane, the name of the procedure pointed to by the blue current line pointer is displayed.

To	Do this
Browse a list of all the procedures in the displayed file	Click the arrow on the right side of the Procedure Locator and select Browse procedures in current file from the list that appears. The Procedures: <i>current_file</i> window opens. To display a procedure in the source pane, click the name of the procedure in the list.
Browse a list of all the procedures in your program	Do one of the following: • In the command pane, enter the browse procs command. For information about the browse command, see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. • Click the arrow on the right side of the Procedure Locator and select Browse procedures in program from the list that appears. • Select Browse → Procedures. Performing any of the above actions opens a Procedures Browse window. To display a procedure in the source pane, click the name of the procedure in the list.

For more information about browsing procedures, see “Browsing Procedures” on page 200.

Using Navigation History Buttons

The Debugger keeps a history of the source locations you have viewed. You can use the **Back** (↶) and **Forward** (↷) buttons to display source code you have recently viewed in the source pane. These buttons function similarly to the back and forward buttons found in Web browsers. If you click and hold either of these buttons, you can select from a short list of previously visited source locations.

See also the **indexnext** and **indexprev** commands in Chapter 12, “Navigation Command Reference” in the *MULTI: Debugging Command Reference* book.

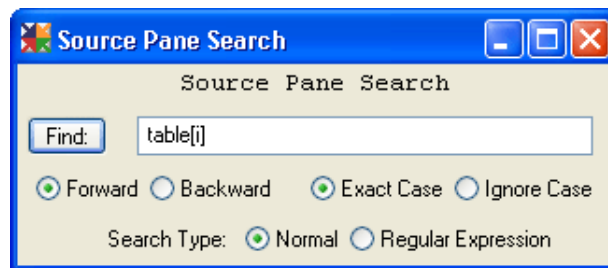
Using the Source Pane Search Dialog Box

The **Source Pane Search** dialog box provides a simple way for you to search your code quickly and easily. To open the **Source Pane Search** dialog box, do one of the following.

- In the command pane, enter the **dialogsearch** command. For information about the **dialogsearch** command, see Chapter 16, “Search Command Reference” in the *MULTI: Debugging Command Reference* book.
- From the main Debugger window, select **Tools → Search**.
- Use the Ctrl + Shift + F keyboard shortcut.

To search:

- Enter the text you are searching for in the **Find** field.



- Use the radio buttons to specify the direction of the search (**Forward** or **Backward**), the case sensitivity of the search (**Exact Case** or **Ignore Case**), and the type of search (**Normal** or **Regular Expression**). The default settings are **Forward**, **Exact Case**, and **Normal**.
- Click **Find**.

MULTI highlights the first match of the search string in the source pane. To find the next match, click **Find** again. The search wraps at the end of the source file.



Note You can also search the source pane incrementally by using simple keyboard shortcuts (see “Incremental Searching” on page 130) or by entering the **fsearch** or **bsearch** commands in the command pane. For information about these and about other commands that allow you to search from the command pane, see Chapter 16, “Search Command Reference” in the *MULTI: Debugging Command Reference* book.

Viewing Program and Target Information

In addition to viewing information in the source pane, you can use stand-alone windows to view more detailed information about your program and the state of your target during program execution. The following sections provide a brief introduction to the types of information you can view, and in some cases, edit.

Viewing Information in Stand-Alone Windows

The MULTI Debugger includes view windows you can use to examine aspects of your source code, debugging information, and target memory information. The following table briefly describes these windows and how to open them from the main Debugger window. For information about the main Debugger window, see Chapter 3, “The Main Debugger Window”. Each window listed in the following table is described in more detail later in this book.

For general information about navigating MULTI Debugger windows, including information about how to use multiple windows and how to use Debugger commands to control and update windows, see “Navigating and Searching Basics” on page 130.

\$locals\$ window

Displays information about variables local to the current function (that is, the function where the program counter (PC) pointer is located). If the PC pointer moves to a new function, the content of the **\$locals\$** window changes to display local variables for that function. For information about the PC pointer, see “The Source Pane” on page 39.

To open this window:

- Click the **Locals** button ().

The **\$locals\$** window is a specialized Data Explorer window. For information about the Data Explorer, see Chapter 9, “Viewing and Modifying Variables with the Data Explorer”.

Breakpoints window

Displays and allows you to configure software breakpoints, hardware breakpoints, and tracepoints.

To open this window, do one of the following:

- Click the **Breakpoints** button (.
- Select **View** → **Breakpoints**.
- Enter the **bpview** command in the Debugger command pane. For information about this command, see Chapter 4, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

For more information about the contents and features of this window, see “The Breakpoints Window” on page 123.

Browse window

Allows you to browse information about procedures, global variables, source files, data types, and cross references.

To open this window:

- Select **Procedures**, **Globals**, **Files**, or **All Types** from the **Browse** menu.

For more information, see “The Browse Window” on page 194.

Call Stack window

Lists a program's functions in the order in which they are called. In addition to naming each function, this window provides the name and value of each argument.

To open the **Call Stack** window, do one of the following:

- Click the **Call Stack** button ().
- Select **View** → **Call Stack**.
- Enter the **callsvi** command in the Debugger command pane. For information about this command, see Chapter 6, "Call Stack Command Reference" in the *MULTI: Debugging Command Reference* book.

For more information, see "Viewing Call Stacks" on page 376.

Data Explorer window

Displays a variable, its type, and its current value.

To open a Data Explorer window:

- Double-click a variable in the source pane. The variable may be of any type, whether an integer, structure, array, or class. If the Data Explorer contains a pointer, double-clicking the pointer opens a new Data Explorer window and displays the variable pointed to. This feature makes it easy to follow linked lists and other complex data structures.

MULTI updates the Data Explorer to show value changes every time your process stops. However, you can freeze Data Explorer windows to save a view of values at a given point and later compare them with values at subsequent points in the process.

For more information about the Data Explorer window, see Chapter 9, "Viewing and Modifying Variables with the Data Explorer". For more information about viewing variables, see "Viewing Variables" on page 277.

Graph View window

Displays an include file dependency graph.

To open the Graph View window:

- Select **Browse** → **Includes**.

For more information, see "The Graph View Window" on page 227.

Memory Test Wizard

Helps you perform memory test operations.

To open this window:

- Select **Target** → **Memory Test**.

For more information, see Chapter 17, "Testing Target Memory".

Memory View window

Displays memory content information in various formats.

To open this window, do one of the following:

- Click the **Memory** button ()
- Select **View** → **Memory**.
- Enter the **memview** command in the Debugger command pane. For information about this command, see “General Memory Commands” in Chapter 11, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.
- In a Data Explorer, select **Tools** → **Memory View on “variable”**.

For more information, see Chapter 13, “Using the Memory View Window”.

MULTI Editor

Allows you to modify source files.

To use the Editor to view and edit source code for a function:

- Double-click the function name in the Debugger’s source pane. The MULTI Editor opens on the function. (If you set another editor as your default, that editor opens on the function.)

For more information about the MULTI Editor, see the *MULTI: Editing Files and Configuring the IDE* book.

Note Browser window

Allows you to associate Debugger Notes with any line of source or assembly code.

To open this window, do one of the following:

- Select **View** → **Debugger Notes**.
- Enter the **noteview** command in the Debugger command pane. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

For more information, see Chapter 8, “Using Debugger Notes”.

OSA Explorer

Displays information about a multitasking operating system.

To open this window, do one of the following:

- Select **View** → **OSA Explorer**.
- Enter the **osaexplorer** command in the Debugger command pane. For information about this command, see “Object Structure Awareness (OSA) Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.
- In the Task Manager window, click the **OSA Explorer** button ().
- In the Task Manager window, select **View** → **OSA Explorer**.

For more information, see “The OSA Explorer” on page 452.

Process Viewer

UNIX only

Lists processes on your target.

To open this window:

- Enter the **top** command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

Profile window

Reports performance, function, graph, and coverage analysis profiling information.

To open this window, do one of the following:

- Select **View** → **Profile**.
- Enter the **profile** command in the Debugger command pane. For information about this command, see Chapter 13, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.

For more information, see “The Profile Window” on page 348.

Register Information window

Displays detailed information, including bitfields and documentation, about a specific register.

To open this window, do one of the following:

- In the Debugger command pane, enter the **regview** command followed by a register name. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.
- In the **Register View** window, double-click a register with a bitfield definition.
- In the **Register View** window, right-click a register name and select **Open an Info View on *register_name*** from the menu that appears.

For more information, see “The Register Information Window” on page 250.

Register View window

Displays current register content.

To open this window, do one of the following:

- Click the **Registers** button ()
- Select **View** → **Registers**.
- Enter the **regview** command in the Debugger command pane. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

For more information, see Chapter 11, “Using the Register Explorer”.

Serial Connection Chooser window

Allows you to interact with a serial terminal.

To open this window, do one of the following:

- Select **Tools** → **Serial Terminal** → **Make Serial Connection**.
- Run the **mterminal** command from the command line (not from the Debugger command pane).

For more information, see Chapter 20, “Establishing Serial Connections”.

Task Manager for *target* window

Debug servers that support multitasking debugging only

Displays the current status of tasks running on an operating system.

To open this window, do one of the following:

- Select **View** → **Task Manager**.
- Enter the **taskwindow** command in the Debugger command pane. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.
- In the **Connection Organizer**, select **Target** → **Show Task Manager**.

For more information, see “The Task Manager” on page 424.

Tree Browser

Displays information about classes, static calls, and file calls in the target program.

To open this window:

- Select **Browse** → **Classes**.
- Select **Browse** → **Static Calls**.
- Select **Browse** → **File Calls**.

For more information, see “The Tree Browser Window” on page 220.

Viewing Information in the Command Pane

In addition to using the specialized, stand-alone windows described in the previous section to view program information, you can also view some information in the command pane of the main Debugger window, as described next.

- To view information about a variable, pointer, or structure, click its name in the source pane. When you click an item in the source pane, the resulting selection is evaluated as an expression by the **MULTI examine** command, which may perform macro expansion in its evaluation. For information about the **examine** command, see Chapter 9, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book. See also Chapter 12, “Using Expressions, Variables, and Procedure Calls”.

For information about viewing variables, see “Viewing Variables” on page 277. When you click a pointer, the command pane also displays the value of the object pointed to. When you click a structure, the command pane displays the whole structure, with every structure or class element labeled.

- To evaluate an expression, type it into the command pane. For more information about evaluating and working with expressions, see Chapter 12, “Using Expressions, Variables, and Procedure Calls”.



Tip You can also use Debugger commands to print a variety of program and target information to the command pane. For detailed information, see Chapter 9, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book.

Using Debugger Notes

Chapter 8

This Chapter Contains:

- Creating and Editing Debugger Notes
- Organizing Debugger Notes Into Groups
- Viewing Debugger Notes

This chapter describes how to create, edit, and access Debugger Notes.

Debugger Notes allow you to associate notes with any line of source or assembly code. You can quickly jump to any part of your program where you have set a Note. It is also possible to use breakpoints to mark parts of your program that are of particular interest. Debugger Notes, however, have the following advantages over breakpoints when marking locations:

- You can set Debugger Notes on source lines that do not have code associated with them, such as comments.
- You can simultaneously edit multiple Debugger Notes.
- You can organize Debugger Notes into groups.
- If you rebuild your program, Debugger Notes are not deleted—even when they are no longer in a valid location. In addition, if the line of source code where you set the Debugger Note moves, the Debugger Note relocates automatically.
- You can quickly display and navigate to any Debugger Note by pressing the F12 key.

Creating and Editing Debugger Notes

Debugger Notes can contain any kind of text and can be associated with any line of code. The line numbers of lines that contain Debugger Notes are shaded gray.

To set a Debugger Note:

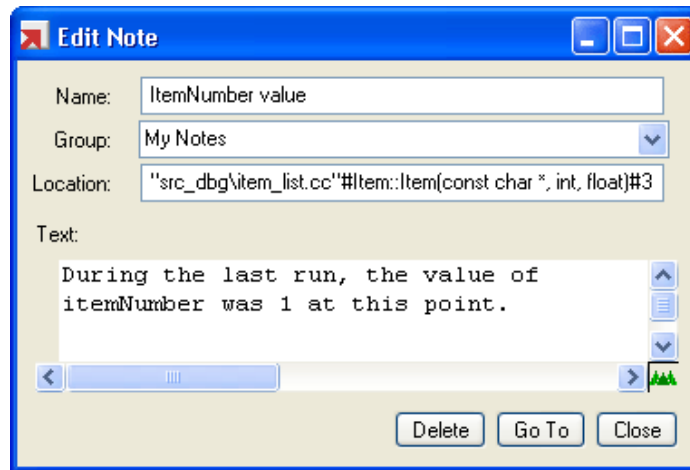
- Right-click a line in the source pane and select **Create Note**, or
- Enter the **noteedit** command in the Debugger command pane. For information about this command, see Chapter 8, “Debugger Note Command Reference” in the *MULTI: Debugging Command Reference* book.

Editing a Single Debugger Note

You can modify a single pre-existing Debugger Note with the **Edit Note** window. To open this window, do one of the following:

- Double-click the shaded gray area in the line number column.
- Right-click the shaded gray area and select **Edit Note**.

- Enter the **noteedit** command in the Debugger command pane. For more information about this command, see Chapter 8, “Debugger Note Command Reference” in the *MULTI: Debugging Command Reference* book.



The following table explains each property you can set from this window.

Name	Assigns a name to the Debugger Note. The Note Browser and Note listings display this name. If you do not specify a name when you create a new Note, MULTI uses a brief version of the Note location as the default name.
Group	Specifies the name of the group where the Debugger Note is located. A Debugger Note must always exist in exactly one group. To put the Note in a group that already exists, select a group from the drop-down list. To create a new group, type the new group name in the Group field.
Location	Specifies the location of the Debugger Note. This may be either an address expression or a source line. For information about using an address expression to specify a location, see “Using Address Expressions in Debugger Commands” in Chapter 2, “Using Debugger Commands” in the <i>MULTI: Debugging Command Reference</i> book.
Text	Assigns text to the Note.

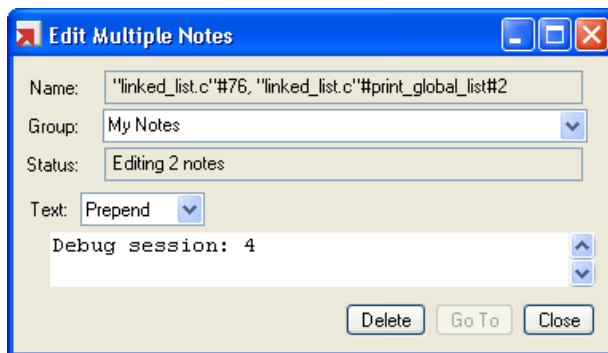
The buttons in this window allow you to perform the actions listed in the following table.

Delete	Deletes the Note displayed in the window. A dialog box asking you to confirm the deletion appears.
Go To	Navigates to the displayed Note.
Close	Closes the window, saving any changes you made to the Note. If the Note location is not valid, a warning message appears.

Editing Multiple Debugger Notes

You can select and modify multiple Notes with the **Edit Multiple Notes** window. This window provides a quick way to simultaneously edit the group or text of multiple Notes. To open this window, perform the following steps:

1. Select **View** → **Debugger Notes** or enter the **noteview** command to open the **Note Browser**. For information about the **noteview** command, see Chapter 8, “Debugger Note Command Reference” in the *MULTI: Debugging Command Reference* book.
2. Select **List** from the **Style** drop-down box.
3. Highlight more than one Debugger Note, right-click, and select **Edit** from the menu that appears.



In the **Edit Multiple Notes** window, use the **Text** drop-down list to set whether text should be appended or prepended to the selected Notes. Enter the desired text in the field underneath the **Text** drop-down. When you click the **Close** button, the text you entered is added to the text of all the Notes listed in the **Name** field.

Removing Debugger Notes

To delete a Note, do one of the following:

- Enter the **notedel** command followed by an address expression or the number of the Debugger Note. For information about the **notedel** command, see Chapter 8, “Debugger Note Command Reference” in the *MULTI: Debugging Command Reference* book.
- Right-click the shaded area and select **Remove Note**.

Organizing Debugger Notes Into Groups

You can organize Debugger Notes into groups to make it easier to work with a large number of them. A Debugger Note group is simply a named collection of Debugger Notes. You can view Note groups in the **Note Browser**, and you can easily delete all Notes in a group or move them to another group. For information about the **Note Browser**, see “Viewing Debugger Notes” on page 157.

Each Debugger Note you create always exists in exactly one group. If no groups exist when you create a Note, MULTI creates a new group called `<default>` to hold the Note. This group becomes the *active group*, which indicates that any Note you create is put into this group by default.

To add a Note to a group that is not the active group, you must either:

- Change the active group by clicking  in the **Note Browser**, or
- Enter the command **noteedit -group group_name**. For information about the **noteedit** command, see Chapter 8, “Debugger Note Command Reference” in the *MULTI: Debugging Command Reference* book.

Viewing Debugger Notes

To display the text stored in a Note, hover over the shaded area; the text appears in a tooltip. Alternatively, single-click the shaded line number area, and the text appears in the command pane.

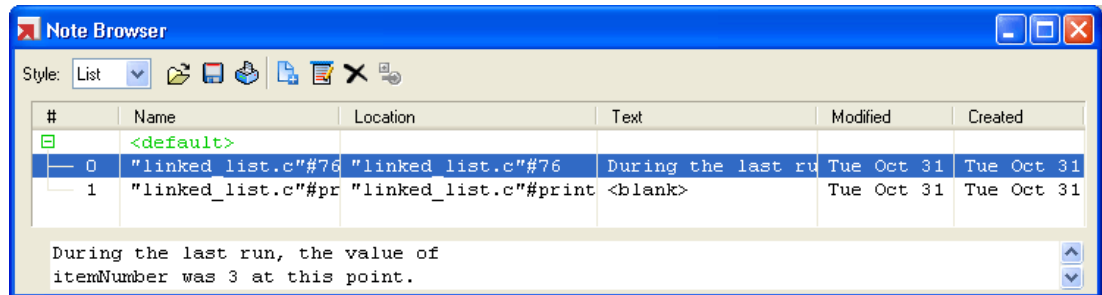
To list all your program’s Debugger Notes in the command pane, enter the **notelist** command. To list all your program’s Debugger Notes in the **Note Browser**, select **View → Debugger Notes** or enter the **noteview** command. For information about these commands, see Chapter 8, “Debugger Note Command Reference” in the *MULTI: Debugging Command Reference* book.

The next section contains details about the **Note Browser**.

The Note Browser

The **Note Browser** displays your Notes in a list or in full report style.

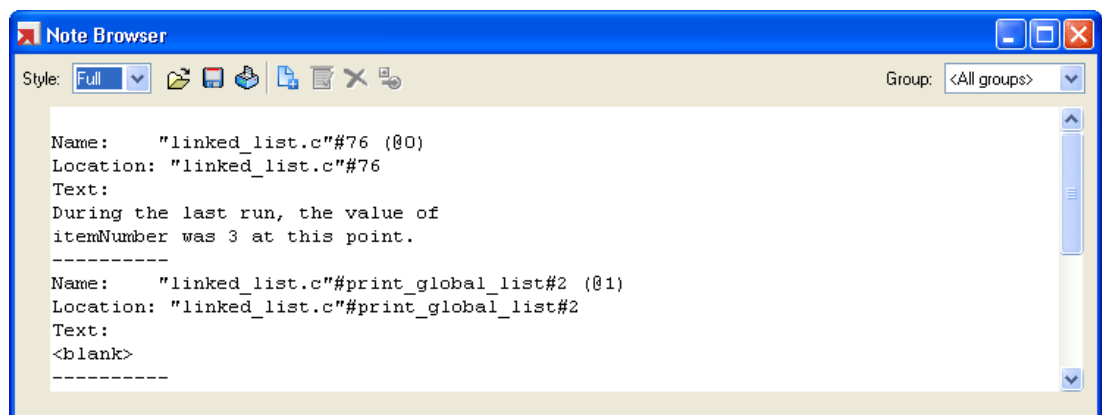
The list style display presents all Debugger Notes in a list format in which you can expand or collapse groups.



To view the list style display, select **List** from the **Style** drop-down box. To edit multiple Notes and to directly modify a group name, use the list style display. For more information, see “Editing Multiple Debugger Notes” on page 156.

In this display, MULTI highlights the active group in green to indicate that it adds Notes to this group by default. Clicking a Note navigates to that Note in the Debugger window.

The full report style display presents all Debugger Notes in report format. Full style display shows the Note’s name, location, and text. This display allows you to search easily through Note text by using the incremental search capability (Ctrl+F and Ctrl+B). For more information, see “Incremental Searching” on page 130.



To view the full style display, select **Full** from the **Style** drop-down box.

In this display, the **Edit** button () is always disabled. The **Group** field, located in the upper-right corner of the window, determines whether the Notes from all groups are displayed or only those Notes from a particular group. The **Group** field also determines whether the **Set Active Group** button () is enabled. Choose from the **Group** drop-down list according to your preferences.

The following table describes the buttons available in the **Note Browser** window.

Button	Action
	Opens a previously saved Debugger Note list from the file you choose. For more information, see the notestate command in Chapter 8, “Debugger Note Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
	Saves the current list of Debugger Notes to the file you choose. For more information, see the notestate command in Chapter 8, “Debugger Note Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
	Prints the current view.
	Creates a new Debugger Note at the line selected in the Debugger window.
	Allows you to edit the selected Note(s) or group. This button is disabled if you select both a Note and a group. It is also disabled in full style display. For more information, see the noteedit command in Chapter 8, “Debugger Note Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
	Deletes the selected Notes and/or groups. For more information, see the notedel command in Chapter 8, “Debugger Note Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
	Sets the active group, to which Notes are added by default if you do not specify a group. This option is disabled if you select the active group.

Viewing Debugging Information and Program Details

Part II

Viewing and Modifying Variables with the Data Explorer

Chapter 9

This Chapter Contains:

- Opening a Data Explorer
- The Data Explorer Window
- Viewing Multiple Items in a Data Explorer
- Updating Data Explorer Variables
- Freezing Data Explorer Variables
- Types of Variable Displays in a Data Explorer
- Changing How Variables are Displayed in a Data Explorer
- Modifying Variables from a Data Explorer
- Configuring Data Explorers
- Data Explorer Messages
- Data Explorer Menus

The Data Explorer is a graphical tool that displays variables of any type in a number of different formats. You can modify the values of variables from a Data Explorer.

Opening a Data Explorer

To open a Data Explorer, do one of the following:

- Double-click a variable in the Debugger source pane.
- Double-click a variable in a preexisting Data Explorer. This opens a new Data Explorer on the double-clicked variable and leaves the original Data Explorer open and unchanged.
- Enter the **view** command in the Debugger’s command pane, where:
 - **view** *expr* [, *expr*]... — Opens a Data Explorer that displays the specified expression(s) *expr*.
 - **view** *type* — Opens a Data Explorer that displays the type *type*.
 - **view** **address* — Opens a Data Explorer that displays the contents of the given location in memory. You must enter an asterisk (*) before the address name.
 - **view** \$locals\$ — Opens a Data Explorer that displays all local variables. This command is equivalent to the **localsview** command and the **Locals** button () located in the Debugger. For information about the **localsview** command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.
 - **view** *number*:\$locals\$ — Opens a Data Explorer that displays local variables for the procedure located *number* levels up the stack.

For more information about the **view** command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

If a Data Explorer is already open when you perform any of the preceding actions, the variables you specified appear in the existing Data Explorer.

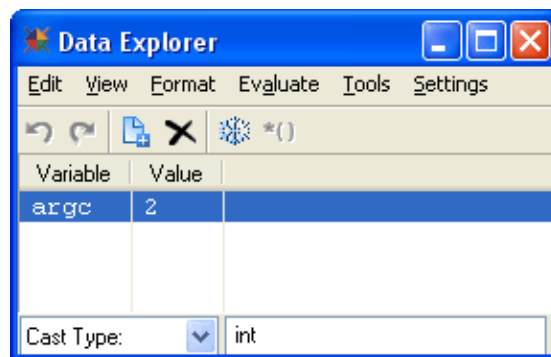
To close a Data Explorer, press Ctrl+Q or select **Edit** → **Close**. Enter the **viewdel** command to close all Data Explorers associated with the active Debugger window. This command also closes Browse, **Register View**,

Memory View, **Call Stack**, and **Breakpoints** windows. For more information about the **viewdel** command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

The Data Explorer Window

The Data Explorer window displays information about your program variables. The format of this information depends on what type of variables you are viewing and on a number of configuration and formatting options you can modify. For information about modifying configuration options, see “Changing How Variables are Displayed in a Data Explorer” on page 175 and “Data Explorer Menus” on page 182.

A sample Data Explorer follows:



By default, the Data Explorer displays two columns. The **Variable** column displays the names of variables, types, members, or expressions in an expandable tree format. Numbers preceding variable names indicate call stack depth. For example, **2: myval** indicates that the variable **myval** is two stack levels up from the current program counter. The **Value** column displays values when applicable.



Note For a description of all menu items available from the Data Explorer, see “Data Explorer Menus” on page 182.

The Data Explorer Toolbar

The Data Explorer toolbar contains the following buttons, from left to right:

- **Undo** () — Undoes your last action. Click **Undo** once for each action you want to undo. The **Undo** button appears dimmed when you reach the point at which no more actions can be undone.

This button is only available for certain operations. The most common operations you can undo are deleting variables from a Data Explorer, making arrays, casting variables to a different type, rerooting members, and dereferencing pointers.

- **Redo** () — Restores the last action you undid. Click **Redo** once for each action you want to restore. The **Redo** button appears dimmed when you reach the point at which no more actions can be restored.

This button is only available after you have undone certain operations. The most common operations you can redo are deleting variables from a Data Explorer, making arrays, casting variables to a different type, rerooting members, and dereferencing pointers.

- **Add Variable** () — Adds a specified variable to the Data Explorer. Specify the new variable in the dialog box that appears.
- **Delete** () — Removes the selected variable from the Data Explorer.
- **Freeze** () — Toggles the selected variable between frozen and unfrozen mode. When the variable is frozen, the Data Explorer does not automatically update it, and you cannot change its value.

Click the **Freeze** button again to reactivate the variable. When the variable is unfrozen, the Data Explorer updates it every time your process stops. For more information, see “Freezing Data Explorer Variables” on page 169.

- **Dereference Pointer** () — Displays the actual value, rather than the address, of the variable a pointer points to. This button is only available if the selected variable is a pointer.

The Edit Bar

The edit bar allows you to change the type and value of a selected variable or define the bounds of a selected array. The edit bar is located at the bottom of the Data Explorer. The left side of the edit bar contains a drop-down menu where you can specify the action you want to perform. Descriptions of each action follow. The right side of the edit bar contains a text box (or two, depending on the action you select) where you can input new values.

- **Cast Type** — Casts the selected variable to the newly specified type. Specify a new type by entering a valid type in the text field. Press Enter to see your change take effect. If you enter an invalid type, your text appears red and no change occurs.

The **Undo** button () becomes available after you change a variable's type. Click the **Undo** button to return to the previous type.

The **Cast Type** entry is automatically selected when you click a variable in the **Variable** column.

- **Edit Value** — Edits the value of the selected variable. Define a new value by entering a valid number, string, or enumeration in the text field. Press Enter to see your change take effect. If you enter an invalid value, your text appears red and no change occurs.

The **Edit Value** entry is automatically selected when you click in the **Value** column.

- **Array Bounds** — Specifies the range of elements to display. In the text boxes, enter the indexes with which you want to begin and end the array. Press Enter to see your changes take effect.

The **Undo** button () becomes available after you change array bounds. Click the **Undo** button to return to the previous view.

The **Array Bounds** entry is automatically selected when you click an array in the **Variable** column.

Viewing Multiple Items in a Data Explorer

You can use the Data Explorer to view multiple variables or to view the same variables at different points during a process.

When you double-click a variable in the source pane or enter the **view** command followed by a variable, MULTI opens a Data Explorer displaying information about that variable. Information about each additional variable you specify—either by double-clicking in the source pane or by entering the **view** command—is added to the existing Data Explorer. For more information about the **view** command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

If you double-click a variable in the source pane more than once or if you enter the **view** command to specify the same variable more than once, the variable appears in the same Data Explorer multiple times. You can freeze one instance of the variable and leave the other unfrozen to display the same variable at different points in the process. For information about freezing variables, see “Freezing Data Explorer Variables” on page 169.

To open a new Data Explorer on a variable listed in an existing Data Explorer, double-click the item in the Data Explorer. The original Data Explorer remains open and unchanged, and a new Data Explorer opens on the double-clicked item. The undo/redo history for the original Data Explorer is not transferred; the new Data Explorer starts with a new history.

You can drag variables from the Debugger’s source pane or from another Data Explorer into an open Data Explorer. If you drag a variable from the source pane, MULTI copies it into the Data Explorer. If you drag a variable from another Data Explorer, MULTI moves the variable into the specified Data Explorer and removes it from its original location.

Updating Data Explorer Variables

The Data Explorer updates the values of its non-frozen variables each time your process stops. You can also force an update of non-frozen variables. In the Debugger command pane, enter the **update** command with no arguments. If necessary, the Data Explorer halts the process to update the data and then resumes the process.

You can also enter the **update interval** command to schedule periodic updates. By updating Data Explorer variables every *interval* seconds while your process is running, this command allows you to monitor the value of variables continually. To deactivate this update, re-enter the **update interval** command with the interval set to zero (0).



Note The **update** command also attempts to refresh other view windows including the **Register View**, **Memory View**, and **Call Stack** windows. For more information about the **update** command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

Freezing Data Explorer Variables

You can freeze Data Explorer variables to compare the values of variables at different times. If you double-click a variable in the source pane more than once or if you enter the **view** command to specify the same variable more than once, the variable appears in the same Data Explorer multiple times (see “Viewing Multiple Items in a Data Explorer” on page 168). You can freeze one instance of the variable and leave the other unfrozen.

To freeze a variable, do one of the following:

- Select the variable and click the **Freeze** button (.
- Select the variable and then select **View** → **Freeze** “variable”.

While the frozen variable is selected, the **Freeze** button () appears to be pushed down and a tick mark appears beside the **Freeze** “variable” menu item. The text for frozen variables is blue. You cannot update or edit frozen variables.

To reactivate a frozen variable, click the **Freeze** button again or select **View** → **Freeze** “variable” again. The Data Explorer updates the value of the variable to

reflect the current state of the process and continues to automatically update it each time the process stops.



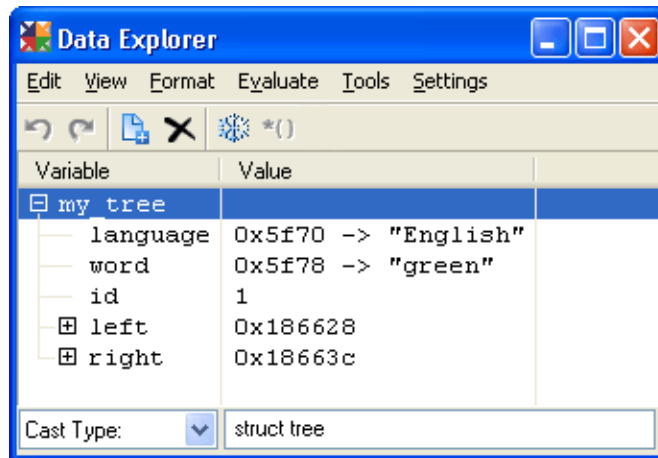
Tip To freeze an original variable in an existing Data Explorer and open the same variable in another Data Explorer, select the variable and press Ctrl + D.

Types of Variable Displays in a Data Explorer

The following sections describe example Data Explorers for structures, linked lists, arrays, and C++ classes.

Displaying Structures

The following graphic displays a Data Explorer showing a structure named `my_tree`. It was generated with the view `my_tree` command.



In this example, the name of the displayed variable is `my_tree`, and its type is `struct tree`. This structure contains five fields, which each appear on separate lines: `language`, `word`, `id`, `left`, and `right`. These fields are in **Natural** format. The Data Explorer dereferences pointers to simple items and shows the value of the item pointed to.

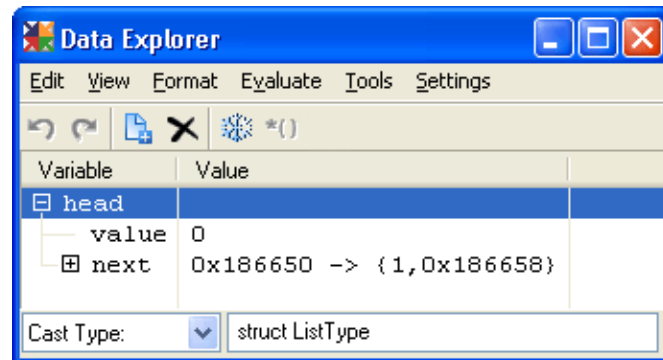


Note In C++, the Data Explorer marks member variables stored by reference with an “@” preceding the address.

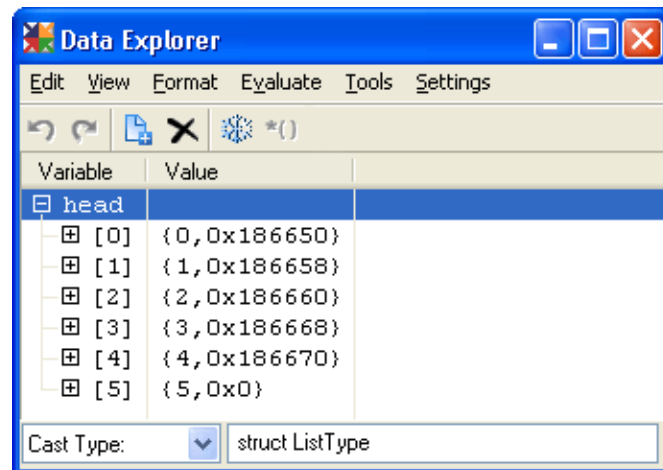
Displaying Linked Lists

The Data Explorer displays linked lists in two different formats. The graphic below represents a typical Data Explorer view for the following structure definition:

```
struct ListType {
    int value;
    struct ListType *next;
}
```



The second format—formatting a linked list as a container of elements—displays the list as if it were an array. As the following graphic demonstrates, this format greatly simplifies the display of these common structures.



To format a linked list as a container of elements, select a member or variable of the type you want to display, and select **View** → **View “variable” as Container**. This menu item is only available if you select a pointer to an aggregate type.

The **Expand As Container** dialog box opens. This dialog box gives you the following choices for how to display your data type:

- **Null-terminated list** — Displays the type as a C-style null-terminated list. A null-terminated list is a linked list that consists of a series of structures connected via *next* pointers and terminated with a NULL pointer. Choose the member that is the *next* pointer in the “**Next**” **Pointer** drop-down list, and click **OK**.
- **Circular list** — Displays the type as a C-style circular list. A C-style circular list is a linked list that consists of a series of structures connected via *next* pointers that form a loop. For the purpose of display, MULTI terminates the list when iteration returns to the initial node. Choose the member that is the *next* pointer in the “**Next**” **Pointer** drop-down list, and click **OK**.
- **Binary tree** — Displays the type as a tree of objects with two children, *left* and *right*. Chose the *left* and *right* pointers from the appropriate drop-down lists, and click **OK**. MULTI traverses the tree in order.

After you specify a visualization for the data type, MULTI displays all variables of that type as a container of elements. To display more element rows, select **View** → **Show More Elements**. To display fewer rows, select **View** → **Show Fewer Elements**. To display all element rows, select **View** → **Show All Elements**.

To stop viewing the type as a container, select a variable of that type in a Data Explorer, and then select **View** → **Stop Viewing as Container**. You may also temporarily disable the capability of viewing the type as a container. To do so, select a variable of that type in the Data Explorer and then select **Format** → **Format Container**, clearing the **Format Container** option.



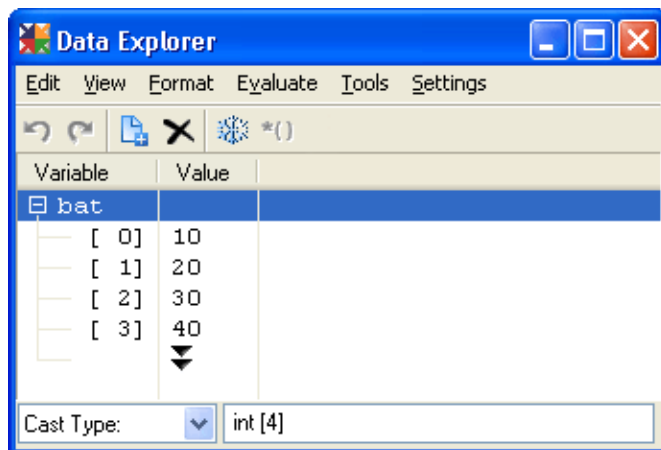
Note In addition to lists, you can also format other containers or structures as containers of elements. For more information, see Appendix D.

You can also use the **viewlist** command to display the elements in a linked list structure. Entering this command is helpful if you are trying to view a few elements in the middle of the linked list. However, the display this command creates often takes up more space than the container display and can be redundant unless you select **View** → **Stop Viewing as Container** first. In addition, you must specify exactly how many elements you want to view, whereas you can simply select the **Show More Elements**, **Show Fewer Elements**, and **Show All Elements** menu items in the container display. For

more information about the **viewlist** command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

Displaying Arrays

The following graphic displays an array in a Data Explorer. It was generated with the `view bat` command, where `bat` is an array of 4 integers.



The Data Explorer shows each element of an array on a separate line. The indices appear in the **Variable** column and the values of each element appear in the **Value** column.

To display a pointer or address type as an array, do one of the following:

- Select a variable in the Data Explorer and then select **Format** → **Make Array**. Each subsequent time you select **Format** → **Make Array**, the size of the array increases by ten (10).
- Select a variable in the Data Explorer and then select **Cast Type** from the edit bar. (If you select the variable from the **Variable** column, **Cast Type** is automatically selected.) Enter any valid type followed immediately by a number enclosed in square brackets. For example, you might enter:

type[*number*], where *type* is a valid type and *number* is any integer.

Press Enter to see your change take effect.

To view a C or C++ character pointer as an array, select the pointer, ensure that **Settings** → **Automatically Dereference Pointers** is enabled (the default), and

then select **Format** → **View Alternate**. You may have to click the plus icon to see individual elements.

To change the size of a selected array, select one of the following:

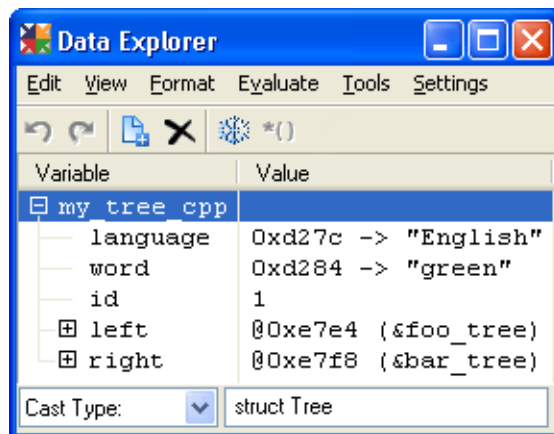
- **View** → **Show More Elements**
- **View** → **Show Fewer Elements**
- **View** → **Show All Elements**

You can also change the size of a selected array by editing the number that appears in square brackets when **Cast Type** is selected in the edit bar.

To change the bounds of a selected array, choose **Array Bounds** from the edit bar. (If you selected the array from the **Variable** column, **Array Bounds** is automatically selected.) Then enter the indices with which you want to begin and end the array. Press Enter to see your change take effect.

Displaying C++ Classes

The following graphic displays a Data Explorer showing C++ classes.



MULTI can display C++ class types, base class types, and virtual base class types in a Data Explorer. The Data Explorer displays static members of a class inside square brackets. It displays members of an anonymous union in an unnamed tree.

Changing How Variables are Displayed in a Data Explorer

You can modify how Data Explorers display variables by doing one of the following:

- Change the type used to display a variable. See “Changing the Type Used to Display a Variable” on page 175.
- View pointers to C++ base classes. See “Viewing Pointers to C++ Base Classes” on page 175.
- Change the base used to display a number. See the descriptions of the **Hex**, **Natural**, **Decimal**, **Binary**, and **Octal** menu items in “The Format Menu” on page 184.

Changing the Type Used to Display a Variable

To change the type used to display a variable in the Data Explorer, select the variable and then select **Cast Type** from the edit bar. (If you select the variable from the **Variable** column, **Cast Type** is automatically selected.) Specify the new type by entering a valid type in the text field. Press Enter to see your variable cast to the newly specified type. Each change you make to the type in this way is independent of previous changes. If you enter an invalid type, your text appears red and no change occurs. You cannot change the type of a frozen variable.

To change a variable’s type to an array of the current type, see “Displaying Arrays” on page 173.

Viewing Pointers to C++ Base Classes

In C++, a Data Explorer can cast a base class pointer to its derived class type. To enable this behavior, select the pointer and then select **Format → Cast To Derived**. With this option enabled, MULTI attempts to find the actual type of an object by matching the name of its virtual table to a known type.

An example follows.

```
#include <iostream>
using namespace std;
class A {
public:
    int a;
    virtual int af() {return a;}
};

class B : public A {
public:
    int b;
};

int main(int argc, char *argv[])
{
    A *ap;
    B b;

    ap = (A*)&b;
    ap->a = 10;

    cout << "ap->a: " << ap->a << endl;

    return 0;
}
```

If, in the preceding example, you open a Data Explorer on the variable `ap` after the statement `ap = (A*)&b;`, MULTI resolves the variable to a pointer to the B class. It does so by matching the virtual function table for B with the one from `ap`. To recast the variable as a pointer to the base class (A), disable **Format → Cast To Derived**.

There are limitations to this capability. The base class must have a virtual function table or no comparison can be made. For example, if class A were defined as follows:

```
class A {
public:
    int a;
};
```

and then you enable **Format → Cast To Derived**, the Data Explorer view does not change because MULTI cannot resolve the derived class. This limitation also applies to multiple inheritance. An example follows.

```
#include <iostream>
using namespace std;
class A {
public:
    int a;
    virtual int af() {return a;}
};

class Z {
public:
    int z;
};

class M : public A, public Z {
public:
    int m;
};

int main(int argc, char *argv[])
{
    A *ap;
    Z *zp;
    M m;

    m.a = 10;

    ap = (A*)&m;
    zp = (Z*)&m;

    cout << "ap->a: " << ap->a << endl;
    cout << "zp->z: " << zp->z << endl;

    return 0;
}
```

In the preceding example, MULTI can determine that the variable `ap` points to an instance of class `M`. However, it cannot determine the actual type that the variable `zp` points to because class `Z` has no virtual function table.

Modifying Variables from a Data Explorer

To modify a variable's value, select the variable and then select **Edit Value** from the edit bar. (If you select the variable's value in the **Value** column, **Edit Value** is automatically selected.) Define the new value by entering a valid number, string, or enumeration in the text field. Press Enter to see your change take effect. If you enter an invalid value, your text appears red and no change occurs.

Configuring Data Explorers

You can configure settings for both individual Data Explorers and for all Data Explorers. The following sections describe how to do so.

Configuring Individual Data Explorer Dimensions

When you first open a Data Explorer, MULTI auto-sizes the window to a size appropriate for the displayed data, but within the minimum and maximum height and width values specified in your current configuration file. MULTI also automatically positions the column divider in the Data Explorer so that the **Variable** column fully displays the longest variable name, type name, member name, or expression.

You can manually resize a Data Explorer and/or column by dragging the window edges or the column divider. However, if you manually resize the window, the Data Explorer no longer auto-sizes to adjust for new data. If you want to use a Data Explorer that automatically resizes, you must close the current Data Explorer and open a new one.

You can also specify global Data Explorer dimensions in the **Data Explorer Options** section of the **Options** dialog box. Select **Config** → **Options** → **Debugger** tab. For more information, see “Configuring Global Data Explorer Dimensions” on page 179.

Setting Global Options for Data Explorers

In addition to formatting individual Data Explorers, you can also set a number of MULTI IDE configuration options that affect the format and behavior of all variables in all Data Explorers.

Limiting the Complexity of Data Displayed

Depending on your target, you may want to minimize the amount of data MULTI reads from the target. MULTI provides configuration options to limit the complexity of information displayed in Data Explorers and to control how much data is read. After making configuration changes, issue the **update** command to see the effects of your changes. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

The following list provides the configuration options.

- To specify a maximum length for the string representation of data in Data Explorers, enter the **configure formatStringMaxLength** *num* command in the Debugger command pane, where *num* is the maximum number of characters. When the accumulated data reaches this length, MULTI does not read any further data from the target unless you select a more detailed view of that data. The minimum value is 1024 bytes. For more information, see the **formatStringMaxLength** option in “Other Debugger Configuration Options” in Chapter 9, “Configuration Options” in the *MULTI: Editing Files and Configuring the IDE* book.
- To specify the maximum number of nested structure levels that Data Explorers display on one line, enter the **configure formatStringMaxDepth** *num* command in the Debugger command pane, where *num* is the maximum depth of the display. Past this maximum depth, the Data Explorer displays values of nested structures as The minimum value is one level. For more information, see the **formatStringMaxDepth** option in “Other Debugger Configuration Options” in Chapter 9, “Configuration Options” in the *MULTI: Editing Files and Configuring the IDE* book.
- To set the maximum initial number of elements in a container, enter the **configure maxContainerDisplaySize** *num* command in the Debugger command pane, where *num* is the maximum number of elements. The default setting is 20. For more information, see the **maxContainerDisplaySize** option in “The More Debugger Options Dialog” in Chapter 9, “Configuration Options” in the *MULTI: Editing Files and Configuring the IDE* book.
- To set the number of elements added to a display when you expand a container, enter the **configure containerSizeincrement** *num* command in the Debugger command pane, where *num* is the number of elements to be added. The default setting is 10. For more information, see the **containerSizeincrement** option in “The More Debugger Options Dialog” in Chapter 9, “Configuration Options” in the *MULTI: Editing Files and Configuring the IDE* book.

Configuring Global Data Explorer Dimensions

Two global configuration options affect the default dimensions of Data Explorers, and one option affects the default positioning of Data Explorers. To set these options, select **Config** → **Options** → **Debugger** tab, or enter the **configure** Debugger command followed by one of the arguments given in the

following table. (For information about the **configure** command, see “Using the configure Command” in Chapter 8, “Configuring and Customizing MULTI” in the *MULTI: Editing Files and Configuring the IDE* book.) The table lists the options available under the **Data Explorer Options** section of the **Debugger** tab and provides the corresponding command arguments.

Minimum initial size (WxH) Specifies the minimum initial size of a Data Explorer in <i>width</i> characters by <i>height</i> lines. The default is 40x3. If you leave this field blank, MULTI auto-sizes the Data Explorers. The equivalent command argument is: • MinViewSize <i>widthxheight</i>
Maximum initial size (WxH) Specifies the maximum initial size of a Data Explorer in <i>width</i> characters by <i>height</i> lines. The default is 40x40. If the minviewsize value you enter is greater in either dimension than the maxviewsize value, the maxviewsize value is used. The equivalent command argument is: • MaxViewSize <i>widthxheight</i>
Initial position (XxY) Specifies the Data Explorer’s initial position from the top-left of the screen. Specify the position in <i>width</i> characters by <i>height</i> lines. The default is 0x0. To give the coordinates in pixels, add a p after them (for example, 100x100p). This option only applies to the first Data Explorer. To avoid overlap, subsequent Data Explorers are offset from the first one that was created. If this option is left unspecified (blank), the Debugger auto-positions all Data Explorers. The equivalent command argument is: • FirstPosition <i>horizxvert</i>[p]

For more information about setting configuration options, see “Setting Configuration Options” in Chapter 8, “Configuring and Customizing MULTI” in the *MULTI: Editing Files and Configuring the IDE* book.

Data Explorer Messages

The Data Explorer may display any of the following messages.

Message	Meaning
...	There is too much data to show on this line. To expand the data in the current window, click the plus sign (+) that appears to the left of the variable. To show the data in a new Data Explorer, double-click the line.
Dead	The variable no longer exists in the current lexical scope. This usually happens when you have stepped past the last use of the variable. If the variable is assigned to a register, the Data Explorer shows the current value of the register along with this message, but the value is most likely meaningless.
Empty	The container has no elements to display.
NaN	Short for “not a number.” The value is not a legal representation of any number. This message applies to floating-point variable types.
No process	No process is currently being debugged, so the Data Explorer cannot display a value.
No symbols for this procedure	Debug symbol information does not exist for the current procedure. This message applies to local variables.
Optimized away	The variable does not exist because a compiler optimization removed it.
Original procedure not on stack	The original procedure, in which the variable was in scope, is no longer on the call stack. The Data Explorer only displays this message when Evaluate → In Context is enabled. For information about Evaluate → In Context , see “The Evaluate Menu” on page 187.
Process running	The process you are debugging is running, so the current value of the variable is unknown.
Not Initialized	The variable has not been assigned an initial value and could have a random value in memory. The Data Explorer shows the current value of the variable, but this value is most likely meaningless.
Unreadable memory	MULTI does not have read access to the memory location that a pointer or address type points to, or the memory location does not exist.

Data Explorer Menus

The following sections describe all menu items available from the Data Explorer. Some menu items are context sensitive and are therefore unavailable in certain situations. Some of these menu items are also available via the Data Explorer's right-click menu.

If a toggle menu item is enabled, a check mark (Windows) or a dot (UNIX) appears to the left of the menu item. Unless otherwise stated, all descriptions of toggle menu items explain the behavior of the item when enabled.

The Edit Menu

The following table describes the menu items available from the Data Explorer's **Edit** menu.

Menu Item	Meaning
Add Variable	Adds a variable to the Data Explorer.
Remove "variable"	Removes the selected variable from the Data Explorer.
Move "variable" to New Window	Moves the selected variable from its original location in an existing Data Explorer to a new Data Explorer.
Copy "variable" to New Window	Copies the selected variable to a new Data Explorer.
Close	Closes the Data Explorer. Equivalent to Ctrl + Q.

The View Menu

The following table describes the menu items available from the Data Explorer's **View** menu.

Menu Item	Meaning
Show	Opens a submenu that lists the following menu items: • Address (Hotkey: S) — Displays the Address column, which shows the addresses of the variables you are viewing. If you are viewing a structure, the field offsets also appear. By default, this item is disabled. • Type (Hotkey: T) — Displays the Type column, which shows the types of the variables you are viewing. By default, this item is disabled. • Variable — Displays the Variable column, which shows the names of the variables you are viewing. By default, this item is enabled. • Value — Displays the Value column, which shows the values of the variables you are viewing. By default, this item is enabled.
Toolbar	Shows the toolbar. By default, this item is enabled.
Freeze “<i>variable</i>”	Freezes the selected variable. When the variable is frozen, the Data Explorer does not automatically update it, and you cannot change its display format or value. When the variable is unfrozen, the Data Explorer updates it every time your process stops. By default, this item is disabled. For more information, see “Freezing Data Explorer Variables” on page 169.
Show Register Info for “<i>variable</i>”	Opens the Register Information window. For more information, see “The Register Information Window” on page 250. This option is only available for local variables stored in a register.
View “<i>variable</i>” as Register	Displays <i>variable</i> as a register, or opens the Register Setup dialog box if <i>variable</i> does not match an existing register definition.
View “<i>variable</i>” as Container	Opens the Expand As Container dialog, which allows you to specify how to display variables of <i>variable</i> 's type as containers. This option is only available for aggregate types or pointers to aggregate types.
Stop Viewing as Container	Stops viewing variables of <i>variable</i> 's type as containers. This option only applies to containers previously displayed with View “<i>variable</i>” as Container (preceding).

Menu Item	Meaning
Show More Elements	Displays 10 more rows of information in the Data Explorer. This menu item is only available if you select the name or address of an array, list, or container. Note: This option can be used to extend an array past its end. This causes target memory to be read past the end of the array and changes the size of the array for the purposes of the Data Explorer.
Show Fewer Elements	Displays 10 fewer rows of information in the Data Explorer. This menu item is only available if you select the name or address of an array, list, or container.
Show All Elements	Displays up to 10,000 element rows of information in the Data Explorer. This menu item is only available if you select the name or address of a statically sized array, list, or container (but not a C pointer).

The Format Menu

The following table describes the menu items available from the Data Explorer's **Format** menu.



Note The first five menu items, which control how numbers are displayed in a Data Explorer, pertain to all types except character pointer types. These string types are always displayed as quoted strings unless the menu items **View Alternate** and **Settings** → **Automatically Dereference Pointers** are enabled, in which case they are displayed as an array of characters.

Menu Item	Meaning
Hex	Displays the numbers and characters of the selected variable, the selected variable's children, etc. in base 16. By default, this item is disabled. However, you can make it the default format for numbers and characters by selecting Config → Options → Debugger tab and checking Display all numbers/characters as hex . See also the note that precedes this table. Equivalent to the hotkey H.

Menu Item	Meaning
Natural	Displays addresses of the selected variable, the selected variable's children, etc. in hexadecimal notation, characters in ASCII notation, and all other numbers in decimal notation. By default, this item is enabled. See also the note that precedes this table. Equivalent to the hotkey N.
Decimal	Displays the numbers and characters of the selected variable, the selected variable's children, etc. in base 10. By default, this item is disabled. See also the note that precedes this table. Equivalent to the hotkey D.
Binary	Displays the numbers and characters of the selected variable, the selected variable's children, etc. in base 2. By default, this item is disabled. See also the note that precedes this table. Equivalent to the hotkey B.
Octal	Displays the numbers and characters of the selected variable, the selected variable's children, etc. in base 8. By default, this item is disabled. See also the note that precedes this table. Equivalent to the hotkey O.
Pad Hex Values	Inserts zeros to the left of hexadecimal values. When you enable this item, the hexadecimal value has the same bit width as the displayed data. By default, this item is disabled. Equivalent to the hotkey X.
Make Array	Displays variables as arrays. For pointer and address types, this treats the pointer as an array such that the first element of the array is located at the address pointed to. For non-pointer types, the Data Explorer displays an array of the appropriate type starting at the variable's location. Each subsequent time you select Make Array, the size of the array increases by ten (10). C and C++ character pointer types must be in View Alternate mode to be viewed as arrays. Additionally, Settings → Automatically Dereference Pointers must be enabled (the default). See View Alternate (following). Equivalent to the hotkey A.

Menu Item	Meaning
View Alternate	Displays data in an alternate way. For most types, the value is displayed in the currently selected format as well as in the alternate format. For example, an integer in natural format is displayed as <code>/decimal/ (/hexadecimal/)</code>. If Settings → Automatically Dereference Pointers is enabled (the default), a character pointer, which is normally displayed as a string, is displayed as an array. All other pointers are treated as integers, which default to displaying in hexadecimal. By default, this item is disabled. Equivalent to the hotkey V.
Cast to Derived	Determines the derived C++ class type of the current object and displays the object cast to that type. This item only applies to the display of C++ classes. For more information, see “Viewing Pointers to C++ Base Classes” on page 175. By default, this item is enabled. Equivalent to the hotkey I.
Show Member Functions	Displays class member functions in type view. By default, this item is enabled. Equivalent to the hotkey F.
Show Template Parameters	Displays the typedefs for template parameters. This option is only valid when you are viewing templated types in C++. By default, this item is disabled. Equivalent to the hotkey W.
Format Container	Displays Standard Template Library (STL) containers as if they were an array of elements. See “Displaying Linked Lists” on page 171. You can also define data descriptions for custom data types. See Appendix D. By default, this item is enabled. Equivalent to the hotkey E.

The Evaluate Menu

The **Evaluate** menu contains four menu items that control the context that is searched to locate the variables being displayed. These options are mutually exclusive; only one of them can be selected at a time. By default, **As Global** is used for all expressions involving only global variables, **By Address** is used if the expression is a static variable, and **In Context** is used for most others. MULTI detects which category the viewed variable falls under, and sets these options accordingly. You can change these options at any time, causing MULTI to re-evaluate the variable with the new setting.

The following table describes the menu items available from the Data Explorer's **Evaluate** menu.

Menu Item	Meaning
In Context	Specifies that when the Data Explorer is updated, MULTI will re-evaluate the selected variable's expression in the same context in which it first evaluated it. For example, if other procedures have been called since the Data Explorer was created, MULTI walks up the stack until it finds a stack frame with the procedure that was executing when the Data Explorer was created. It then evaluates the expression there. If MULTI cannot find such a stack frame, the Data Explorer displays an error. See also As Local (following). Equivalent to the hotkey C.
As Local	Specifies that when the Data Explorer is updated, MULTI will re-evaluate the selected variable's expression within the current procedure at the top of the call stack. See also In Context (preceding). Equivalent to the hotkey L.
As Global	Specifies that when the Data Explorer is updated, MULTI will re-evaluate the selected variable's expression, looking for variables in the global scope and ignoring all procedure scopes. This option is useful when an expression involves only global variables. Equivalent to the hotkey G.
By Address	Specifies that when the Data Explorer is updated, MULTI will use the last valid variable address to display data. MULTI does not re-evaluate the selected variable's expression in any context. This option is useful for examining the contents of local variables in memory before and after they are in scope. Equivalent to the hotkey R.

The Tools Menu

The following table describes the menu items available from the Data Explorer's **Tools** menu.

Menu Item	Meaning
Go to Definition of “<i>variable</i>”	Displays the source code, if available, for <i>variable</i> 's definition in the source pane.
Go to Declaration of “<i>variable</i>”	Displays the source code, if available, for <i>variable</i> 's declaration in the source pane. This is available only for global and static variables.
Browse References of “<i>variable</i>”	Displays <i>variable</i> 's cross references, if any, in a Browse window. For more information, see “Browsing Cross References” on page 216.
Set Watchpoint on “<i>variable</i>”	Sets a watchpoint on <i>variable</i> . For more information, see the watchpoint command in Chapter 4, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
Explore “<i>variable</i>”	Opens a Graph View window on <i>variable</i> . If data descriptions are available for <i>variable</i> 's type, MULTI traverses the data and displays a graph according to the data description. See Appendix D.
Memory View on “<i>variable</i>”	Opens a Memory View window in which you can view and modify memory contents. Initially, this window displays memory at the address of the selected variable. See also the memview command in “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. Equivalent to the hotkey M.
Print	Opens a dialog box that allows you to print the contents of the Data Explorer.

The Settings Menu

The following table describes the menu items available from the Data Explorer's **Settings** menu.

Menu Item	Meaning
Automatically Dereference Pointers	Dereferences pointers viewed in the Data Explorer if the memory they point to seems safe to read. For character pointers, this option reads the string pointed to and displays it in the Data Explorer. Except for character pointer variables, the names of dereferenced variables begin with an asterisk (*). By default, this item is enabled. Equivalent to the configuration option derefPointer. For information about derefPointer, see "The More Debugger Options Dialog" in Chapter 9, "Configuration Options" in the <i>MULTI: Editing Files and Configuring the IDE</i> book.
Color Text Based on Evaluate Context	Applies different colors to local variables, global variables, variables evaluated in context, and variables evaluated by address. By default, this item is disabled.
Default Large Variables to New Window	Opens a new Data Explorer for variables that have more than five rows of information. By default, this item is disabled.

Browsing Program Elements

Chapter 10

This Chapter Contains:

- Browsing Program Elements Overview
- The Browse Window
- The Tree Browser Window
- The Graph View Window

The MULTI Debugger provides several kinds of display windows that allow you to view program elements in different graphical formats. This chapter describes each of the following windows:

- The Browse window — Lists information about the selected object type in a single collapsible and expandable list. For general information about this window, see “The Browse Window” on page 194.
- The Tree Browser window — Displays information about the selected object type in collapsible and expandable lists. This window displays expanded information in new columns, thus offering a hierarchical view of the relationship among the items it displays. For general information about this window, see “The Tree Browser Window” on page 220.
- The Graph View window — Displays an object graph that shows the relationships between items. For general information about this window, see “The Graph View Window” on page 227.

Browsing Program Elements Overview

You can browse the following program elements in one or more of these windows:

- *Procedures* — Select **Browse** → **Procedures**. See “Browsing Procedures” on page 200.
- *Global variables* — Select **Browse** → **Globals**. See “Browsing Global Variables” on page 209.
- *Source files* — See “Browsing Source Files” on page 211.
 - *All* (all source files that contain procedures used in the program you are debugging) — Select **Browse** → **Files**.
 - *Includers* (all source files that directly include the current file) — Right-click an empty spot in the source pane and select **Browse Includers Of This File** from the menu that appears.
 - *Included files* (all files that the current file directly includes) — Right-click an empty spot in the source pane and select **Browse Files Included By This File** from the menu that appears.
 - *Include graph* (a graph of included files) — Select **Browse** → **Includes**. See “Browsing Includes” on page 228.

- *Data types* — Select **Browse** → **All Types**. See “Browsing Data Types” on page 214.
- *Cross references* (for a particular symbol in the program you are debugging) — In the source pane, right-click the symbol and select **Browse References** from the menu that appears. See “Browsing Cross References” on page 216.

Note that this menu option is only available if the program you are debugging was compiled with cross reference information. For more information about how to do this, see the *MULTI: Building Applications* book for your target.

- *Class hierarchies* — Select **Browse** → **Classes**. See “Browsing Classes” on page 224.
- *Calls*:
 - *Static calls by function* (functions that call other functions, and functions that the procedure at the current line pointer calls) — Select **Browse** → **Static Calls**. See “Browsing Static Calls By Function” on page 225.
 - *Dynamic calls by function* (functions that were actually called during run time) — Select **Browse** → **Dynamic Calls**. See “Browsing Dynamic Calls by Function” on page 226.

Note that this menu option is only available if you collected profiling data about function calls (that is, call count data with call graph support enabled). See “Overview of Profiling Methods” on page 341.

- *Static calls by file* (source files whose functions are called from a particular source file) — Select **Browse** → **File Calls**. See “Browsing Static Calls By File” on page 226.



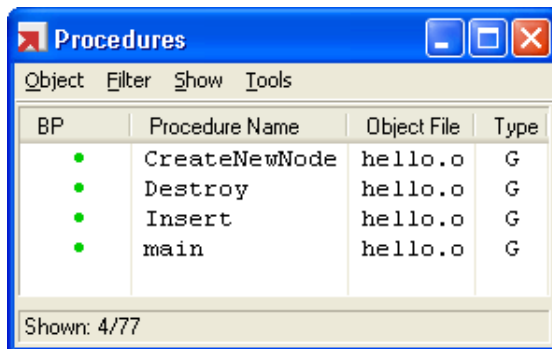
Note For most of these program elements and views, there are alternative ways to open the browsing tools. The following sections document each browsing tool, its views, and the ways you can access it.

The Browse Window

The Browse window is a graphical tool that you can use to view information about procedures, global variables, source files, data types, and cross references. The columns in the main portion of the window vary depending on what type of object you are viewing, but the four menus and the basic behavior of the window are always the same. The following section describes the aspects of the Browse window that remain constant. Later sections document how you can use the Browse window to view specific types of information.

Browse Window Basics

You can open a Browse window in multiple ways—each of which is described in more detail in the following sections. For most of the elements that you can view in a Browse window, you can simply select **Browse** → *program element* from the Debugger. For example, to open a Browse window similar to the one shown below, select **Browse** → **Procedures**.



Note To browse cross references, files that include the current file, or files that the current file includes, right-click in the source pane and select the appropriate menu option from the menu that appears. For more information, see “Browsing Source Files” on page 211 and “Browsing Cross References” on page 216.

As with most MULTI windows, you can:

- Reorder the columns of a Browse window by dragging and dropping column headers,
- Sort the displayed data by clicking the relevant column header,
- Open a context-sensitive shortcut menu by right-clicking in the window.

Every Browse window includes one column known as the *primary column*. Unlike other columns, you can never hide the primary column, but you can change the content formatting via the **Show** menu. The primary column and available formatting options differ based on what type of object you view.

Every Browse window includes the following four menus. Menu options vary according to the type of information you browse.

- **Object** menu — Allows you to switch among viewing procedures, global variables, source files, or data types. The type of program element you are currently viewing has a bullet or check mark next to it. Simply select one of the other choices listed at the top of the **Object** menu to change the type of object being displayed. In a single Browse window, you can only view one type of information at a time, but you can have multiple Browse windows open simultaneously.

This menu also allows you to print the contents of the current Browse window, access MULTI's online help information, or close the Browse window. This menu is exactly the same in all Browse windows.

- **Filter** menu — Allows you to specify filters that limit which objects the Browse window displays. You can set a user-defined filter and/or a selection of predefined filters. The availability of predefined filters varies depending on what type of object you are viewing. For more information about filtering, see "Filtering Content in the Browse Window" on page 196.
- **Show** menu — Allows you to specify the columns displayed in the Browse window and the style of the primary column. The availability of columns and styles varies depending on the type of object you are viewing. Menu options are described in the sections that cover each type of Browse window.
- **Tools** menu — Allows you to perform actions on the information in the Browse window or open other tools that allow you to do so. Most of the options in this menu also appear in the shortcut menu. The availability of menu items in this menu and in the shortcut menu vary depending on what type of object you are viewing. Menu options are described in the sections that cover each type of Browse window.

Each Browse window contains a status bar, which lists the number of items displayed in the format:

Shown: *visible/total*

where:

- *visible* is the number of items that are not filtered. See “Filtering Content in the Browse Window” on page 196.
- *total* (also known as the *base set*) is the number of total objects encapsulated by the Browse window. See “Using Filters” on page 199.

If the Browse window contains contracted headings (see “Browse Window Headings” on page 200), the status bar displays the following alternative format:

Shown: *visible/total (expanded/visible Expanded)*

where:

- *visible* and *total* are the same as above.
- *expanded* is the number of entries that are both visible (i.e., not filtered) and not contracted.

Filtering Content in the Browse Window

In all Browse windows, you can select what data to display by using pre-existing and/or user-defined filters. You can enable and disable filters using the **Filter** menu. Some filters are enabled by default when procedures or global variables are first displayed in a Browse window.

The **Filter** menu displays only those filters that apply to the current type of object. User-defined filters are applied to the names of the objects being shown in the Browse window (see “User-Defined Filters” on page 198). The status box at the bottom of the Browse window lists the number of displayed objects as well as the total number of objects, which includes those that have been filtered out.

Predefined Filters

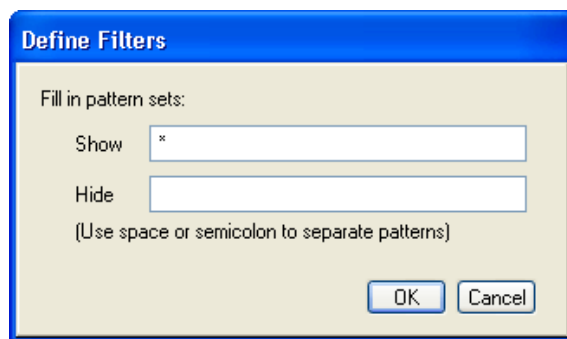
The predefined filters you can enable and disable from the **Filter** menu are described in the following table. Only those filters appropriate to the object type you are viewing are available in the menu. You can apply multiple filters.

Menu option	Effect
Hide C++ VTBLs	Toggles the display of virtual tables in C++ programs. This filter can only be applied to global variables.
Hide C++ Type Identifiers	Toggles the display of type identifiers in C++ programs. This filter can only be applied to global variables.
Hide C++ Type Info	Toggles the display of type information in C++ programs. This filter can only be applied to global variables.
Hide C++ Initialization Names	Toggles the display of initialization names in C++ programs. This filter can only be applied to global variables.
Hide C++ std::*	Toggles the display of objects in the <code>std</code> namespace in C++ programs. This filter can be applied to global variables, procedures, or data types.
Hide .*	Toggles the display of objects whose names begin with a period (.). This filter can be applied to global variables, procedures, or data types.
Hide _*	Toggles the display of objects whose names begin with an underscore (_). This filter can be applied to global variables, procedures, data types, or source files.
Hide Globals from Shared Library	Toggles the display of global variables from shared libraries that were loaded into your program. This filter can only be applied to global variables.
Hide Files without Procedures	Toggles the display of source files that do not contain any procedures. This filter can only be applied to files.
Hide Procedures without Source	Toggles the display of procedures that do not have source code. This filter can only be applied to procedures.
Hide Inlined Procedures	Toggles the display of procedures that are inlined. This filter can only be applied to procedures.
Hide Static Names	Toggles the display of objects that are defined as static. This filter can be applied to either global variables or procedures.
Hide Non-Static Names	Toggles the display of objects that are not defined as static. This filter can be applied to either global variables or procedures.

Menu option	Effect
Hide Writes	Toggles the display of writes. This filter can only be applied to cross references.
Hide Reads	Toggles the display of reads. This filter can only be applied to cross references.
Hide Addresses	Toggles the display of address references. This filter can only be applied to cross references.
Hide Declarations	Toggles the display of declarations. This filter can only be applied to cross references.

User-Defined Filters

To define your own filter to apply to the objects listed in the Browse window, select **Filter** → **User-defined Filter** from the menu bar of any Browse window. The following **Define Filters** dialog box opens:



To specify what objects to display in the Browse window, enter text and wildcard patterns in this dialog box (see “Wildcards” on page 284). To specify multiple patterns in the **Show** and **Hide** text fields, use semicolons or whitespace to separate the patterns.

The next section explains how the Browse window processes user-defined filters with selected predefined filters.

Using Filters

If you specify a user-defined filter and/or one or more predefined filters, the Browse window uses the following algorithm to determine what objects to display.

1. The Browse window determines the base set of objects in the Browse window. When you first open a Browse window, the base set of objects is the set of objects that were initially specified. For example, if you open the Browse window with the **e f*** command (see “Browsing Procedures” on page 200), the base objects are all the procedures whose names begin with the letter **f**. Another example: if you open the Browse window by selecting **Browse** → **Procedures**, the base objects are all the procedures in your program. Each time you change the object type you want to browse by using the **Object** menu, the newly loaded objects become the new base set.
2. The Browse window hides any remaining objects whose names do not match the patterns listed in the **Show** field of the **Define Filters** dialog box.
3. The Browse window hides any remaining objects whose names match the patterns listed in the **Hide** field of the **Define Filters** dialog box.
4. The Browse window hides any remaining objects that match the other filters enabled in the **Filter** menu.

For example, if the user-defined filters are:

- **Show** **f a***
- **Hide** **f a b***

only those objects from the base object set whose names start with **f a** but not **f a b** are displayed.



Note The base set (described above) does not change based on user-defined filters or the selections you make in the **Filter** menu, which only affect what is displayed in the Browse window. However, the base set does change based on your selections in the **Object** menu.

Browse Window Headings

In procedure, user type, and source file Browse windows, displayed data has additional formatting. In each of these cases, items known as *headings* appear with either a plus sign (+) or a minus sign (-) next to their entry in the primary column.

Clicking a plus sign expands the children of that heading; clicking a minus sign contracts the children. The meaning of each of the headings depends on what type of objects are displayed. When you first open a Browse window, all headings appear fully expanded (a minus sign is displayed next to each heading, and all children are visible).

In filtering and sorting, the Browse window treats headings in the same fashion as regular objects. Those headings colored the same as keywords, however, are of a different type than the displayed base object type. If the selected heading is colored like a keyword, certain **Tools** options appear dimmed and certain right-click menu options do not appear.

For example, in the procedure Browse window all headings appear colored like keywords, indicating that they are types and not procedures. As a result, **Tools** → **Browse References** (among others) is unavailable when such an entry is selected. For more specific information about headings in procedure Browse windows, see “Headings in the Procedures Browse Window” on page 207.

Browsing Procedures

You can open a Browse window that displays procedures by selecting a menu option, using a shortcut, or issuing a command. Depending on how you open the Browse window, it displays either all the procedures in a program or some subset of procedures.

- To open a Browse window displaying all procedures in your program, do one of the following:
 - From the Debugger, select **Browse** → **Procedures**.
 - From an existing Browse window that is not currently displaying procedures, select **Object** → **Procedures**.
 - From the Procedure Locator located on the Debugger navigation bar, select **Browse procedures in program**.

- From the command pane, issue the **browse procedures** or **browse procs** command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.
- To open a Browse window displaying only the procedures in a specific file, do one of the following:
 - While viewing the file in the Debugger, select **Browse procedures in current file** from the Procedure Locator on the Debugger navigation bar.
 - Double-click the file in a Browse window displaying source files.
 - From the command pane, issue the **e** command with the arguments `"file" # *` (for example, `e "test.c" # *`). For information about the **e** command, see Chapter 12, “Navigation Command Reference” in the *MULTI: Debugging Command Reference* book.
- To open a Browse window displaying procedures that match a specific pattern, issue the **e** command with a pattern that matches more than one procedure in your program. For example, issuing the **e f*** command opens a Browse window on all the procedures in your program that begin with the letter **f**. For information about the **e** command, see Chapter 12, “Navigation Command Reference” in the *MULTI: Debugging Command Reference* book.
- To open a Browse window displaying the procedures called from a specific procedure or the procedures that call a specific procedure, right-click a procedure in the source pane and select **Browse Other** → **Browse Callees** or **Browse Other** → **Browse Callers**, respectively, from the shortcut menu.



Note Sometimes MULTI opens a modal dialog box version of the Browse window when you have not performed any of the preceding actions. This usually happens when you have issued a command such as **b *** and specified a pattern that matches more than one procedure or an overloaded C++ procedure. You can use this version of the Browse window to specify the procedures you want the issued command to operate on so that the command can continue. For an illustration and instructions on how to use this dialog box, see “Procedure Ambiguities and the Browse Dialog Box” on page 207. For information about the **b** command, see Chapter 4, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

Browsing Procedures General Information

When the Browse window opens, it may contain all the procedures in the program being debugged, or just those meeting all the starting criteria as described above. For example, it may display the procedures that match the wildcard pattern of an **e** command, or the callers of a procedure. But whenever you switch to a different object type with the **Object** menu and then select **Object** → **Procedures** again, the Browse window will display all of your program's procedures.

Procedures are color-coded in the Browse window according to MULTI's color settings. They are displayed in the following way:

- Procedure headings are displayed in the color used to represent keywords (see "Headings in the Procedures Browse Window" on page 207).
- Procedures without source code are displayed in gray.
- Procedures that are inlined are displayed in the color used to represent unused code.
- Procedures that are static (that are not also inlined or lacking source code) are displayed in the color used to represent comments.
- All other procedures are displayed in the normal text color.

Browsing Procedures Show Menu

The following table describes the options available in the **Show** menu when you browse procedures. It also lists which options are enabled by default.



Note The primary column when you browse procedures is the **Procedure Name** column. The formatting options listed below apply only to this column, which is always displayed when you browse procedures.

Option	Default	Meaning
Name	Off	Formats the Procedure Name column (the primary column) so that the use name is displayed.

Option	Default	Meaning
Mangled Name	Off	Formats the Procedure Name column (the primary column) so that the mangled name is displayed. For some languages, this may be the same as what would be displayed under the Name option. For example, in C, the mangled name may be the same as the use name.
Unqualified Name	On	Formats the Procedure Name column (the primary column) so that the unscoped portion of the use name is displayed. For example, in C++, a procedure with the name <code>Foo::bar(int a)</code> would be displayed as <code>bar(int a)</code> . For many entries, this is the same as what would be displayed under the Name option. For more information, see “Headings in the Procedures Browse Window” on page 207.
Breakpoint	On	Shows or hides the BP column. This column displays the first breakdot of the procedure. If a breakpoint is set at the first executable line of the procedure, the icon for the corresponding breakpoint type is shown; otherwise, a green dot is shown. To set a breakpoint at the first executable line of the procedure, click the green dot.
Object File	On	Shows or hides the Object File column. This column displays the object file in which the procedure is located.
Source File	Off	Shows or hides the Source File column. This column displays the source file in which the procedure is located.
Module	Off	Shows or hides the Module column. This column displays the name of the module in which the procedure is located, if any.
Library	Off	Shows or hides the Library column. This column displays the name of the library in which the procedure is located, if any.
Address	Off	Shows or hides the Address column. This column displays the address of the procedure.

Option	Default	Meaning
Size	Off	Shows or hides the Size column. This column displays the size of the procedure in bytes.
Type	On	Shows or hides the Type column. This column displays: • GI if the procedure is an inlined, non-static procedure. • SI if the procedure is an inlined, static procedure. • G if the procedure is a not-inlined, non-static procedure. • S if the procedure is a not-inlined, static procedure.

Browsing Procedures Mouse Operations

The following table describes the results of mouse operations on procedures in the Browse window.

Mouse action	Meaning
Click	Displays the selected procedure in the Debugger's source pane. If you click in the BP column, the Debugger will either insert a breakpoint at the selected procedure if no breakpoint is there, or remove the breakpoint there if one already exists. The breakpoint is set at the first executable line of the procedure.
Double-click	Opens a new window displaying the cross references for the selected procedure.
Right-click	Opens a shortcut menu. For more details, see "Browsing Procedures Tools Menu" on page 205.

Browsing Procedures Tools Menu

The following table describes the options available in the **Tools** menu when you browse procedures. Additionally, the table describes shortcut menu options that become available when you right-click content in the Browse window. The table's "Location" column specifies whether the option appears in the **Tools** menu, the shortcut menu, or both menus.

Option	Location	Meaning
Contract All	Both	Contracts every heading.
Expand All	Both	Expands every heading (this is the way the Browse window looks when first opened).
Global Scope On/Off	Tools menu	Toggles whether global scope is enabled or not. See "Headings in the Procedures Browse Window" on page 207.
Browse References	Both	Displays the selected procedure's cross references (if any) in another Browse window.
Browse Callers	Both	Displays the selected procedure's callers in another Browse window.
Browse Callees	Both	Displays the selected procedure's callees in another Browse window.
Show in Editor	Both	Opens the selected procedure in MULTI's Editor.
Show in Tree Browser	Both	Opens a Tree Browser window to show the selected procedure's static call graph.
Set Breakpoint	Shortcut menu (BP column only)	Sets a software breakpoint. See also the b command in Chapter 4, "Breakpoint Command Reference" in the <i>MULTI: Debugging Command Reference</i> book
Set and Edit Breakpoint	Shortcut menu (BP column only)	Opens the Software Breakpoint Editor window, which allows you to specify and set a software breakpoint. See the b command in Chapter 4, "Breakpoint Command Reference" in the <i>MULTI: Debugging Command Reference</i> book, and also "Creating and Editing Software Breakpoints" on page 106.

Option	Location	Meaning
Set Any Task Breakpoint	Shortcut menu (BP column only)	Sets a software breakpoint which can be hit by any task. This option is available only when connected to a target which supports this type of breakpoint. See the sb command in Chapter 4, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
Edit Breakpoint	Shortcut menu (BP column only)	Opens the Software Breakpoint Editor , which allows you to edit the software breakpoint set on the line.
Remove Breakpoint	Shortcut menu (BP column only)	Removes the software breakpoint from the line.
Enable Breakpoint	Shortcut menu (BP column only)	Enables the software breakpoint located on the line.
Disable Breakpoint	Shortcut menu (BP column only)	Disables the software breakpoint located on the line.
Set Hardware Breakpoint	Shortcut menu (BP column only)	Sets a hardware breakpoint on the line.
Edit Hardware Breakpoint	Shortcut menu (BP column only)	Opens the Hardware Breakpoint Editor , which allows you to edit the hardware breakpoint set on the line.
Remove Hardware Breakpoint	Shortcut menu (BP column only)	Removes the hardware breakpoint from the line.
Enable Hardware Breakpoint	Shortcut menu (BP column only)	Enables the hardware breakpoint located on the line.
Disable Hardware Breakpoint	Shortcut menu (BP column only)	Disables the hardware breakpoint located on the line.
Set Tracepoint	Shortcut menu (BP column only)	Opens the Tracepoint Editor , which allows you to set a tracepoint.
Edit Tracepoint	Shortcut menu (BP column only)	Opens the Tracepoint Editor , which allows you to edit the tracepoint.

Option	Location	Meaning
Remove Tracepoint	Shortcut menu (BP column only)	Removes the tracepoint.
Enable Tracepoint	Shortcut menu (BP column only)	Enables the tracepoint.
Disable Tracepoint	Shortcut menu (BP column only)	Disables the tracepoint.
Set BPs on Entries	Both	Sets breakpoints at the first executable line of all displayed procedures.
Delete BPs on Entries	Both	Deletes breakpoints at the first executable line of all displayed procedures.
Breakpoint Window	Shortcut menu (BP column only)	Opens the Breakpoints window for the program being debugged.

Headings in the Procedures Browse Window

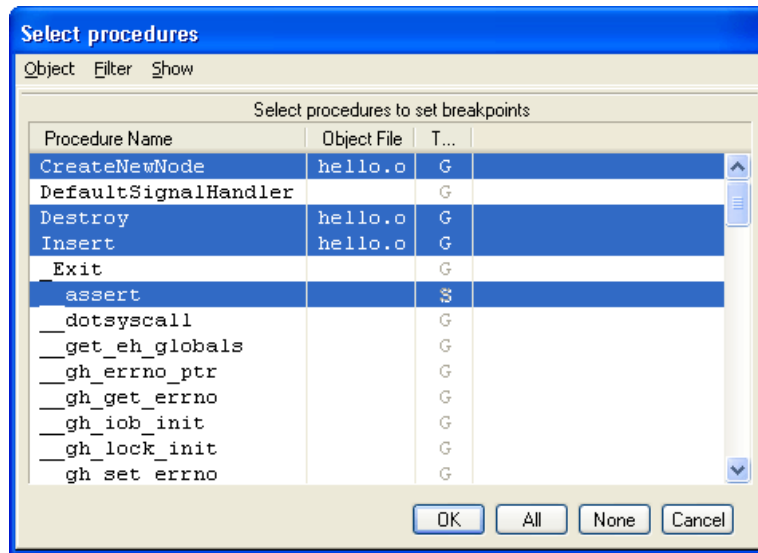
As noted in “Browse Window Headings” on page 200, some entries may be headings when you browse procedures. For procedures, these headings are the “scopes” (or enclosing types) of the procedures listed as children of the heading. For example, in C++ we might have a class `Foo` with a member function `bar(int a)`. The full use name of this function would then be `Foo::bar(int a)`, with `Foo` being the scope and `bar(int a)` the unqualified name of the procedure.

To collect all unscoped types into a global scope, use **Tools → Global Scope On/Off** to enable the global scope option (see “Browsing Procedures Tools Menu” on page 205). This will place all globally accessible functions into a fake scope, to ease viewing by allowing all unscoped names to be hidden by contracting the heading labeled `<global_scope>`.

Procedure Ambiguities and the Browse Dialog Box

Sometimes MULTI will display a modal dialog box version of the Browse window for procedures. This happens if, during a command (such as `b *`), you have specified a pattern that matches more than one procedure or an overloaded

C++ procedure. This results in a procedure name ambiguity, and the command cannot continue. You can use this dialog box to specify which procedure you would like the issued command to operate on, and let the command continue.



By selecting the rows in the Browse dialog box, you can specify which procedures should be used in the action or command to resolve the procedure name ambiguity. Depending on the command, you may be allowed to select several procedures, or just one. After you have made your choice, you can click one of the buttons available at the bottom of the Browse dialog box:

Button	Meaning
OK	Accepts the current selections.
All	Selects all of the procedures displayed in the Browse dialog box, if selecting multiple procedures is applicable.
None	Ensures that none of the procedures displayed in the Browse dialog box is selected, if making no selection is applicable.
Cancel	Closes the Browse dialog box and cancels whatever operation caused the dialog box to appear.

Browsing Global Variables

To open a Browse window for global variables, do one of the following:

- From the Debugger, select **Browse** → **Globals**.
- From the command pane, issue the **browse globals** command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.
- From an existing Browse window, select **Object** → **Globals**.

Browse Globals General Information

Global variables are color-coded in the Browse window according to MULTI’s color settings. They are displayed in the following way:

- Static global variables are displayed in the color for comments.
- All other global variables are displayed in the normal text color.

Browsing Globals Show Menu

The following table describes the options available in the **Show** menu when you browse global variables. It also lists which options are enabled by default.



Note The primary column when you browse global variables is the **Global Name** column. The formatting options listed below apply only to this column, which is always displayed when you browse global variables.

Option	Default	Meaning
Name	On	Formats the Global Name column (the primary column) so that the use name is displayed.
Mangled Name	Off	Formats the Global Name column (the primary column) so that the mangled name is displayed. For some languages, this may be the same as what would be displayed under the Name option. For example, in C, the mangled name may be the same as the use name.
Unqualified Name	Off	Equivalent to the Name option.

Option	Default	Meaning
Object File	Off	Shows or hides the Object File column. This column displays the name of the object file in which the global variable is referred to or defined.
Module	On	Shows or hides the Module column. This column displays the name of the module in which the global variable is located, if any.
Library	Off	Shows or hides the Library column. This column displays the name of the library in which the global variable is located, if any.
Address	Off	Shows or hides the Address column. This column displays the address of the global variable.
Size	Off	Shows or hides the Size column. This column displays the size of the global variable.
Type	On	Shows or hides the Type column. This column displays: • G if the global variable is non-static. • S if the global variable is static.

Browsing Globals Mouse Operations

The following table describes the results of mouse operations on global variables in the Browse window.

Mouse action	Meaning
Click	Displays the selected global variable's definition in the Debugger's source pane, if it can be found.
Double-click	Opens a new window displaying the cross-references for the selected variable.
Right-click	Opens a shortcut menu. For more details, see "Browsing Globals Tools Menu" on page 211.

Browsing Globals Tools Menu

The following table lists the options available in the **Tools** menu when you browse global variables. The same menu is displayed when you right-click content in the Browse window.

Option	Meaning
Print Value	Prints the value of the selected global variable in the command pane. This value is available only if your process is running.
View Value	Opens a Data Explorer to show the value of the selected global variable. This value is available only if your process is running.
Go To Definition	Displays the selected global variable's definition in the Debugger's source pane, if it can be found.
Browse References	Shows the clicked global variable's cross references in a new Browse window.

Browsing Source Files

- To open a Browse window displaying all source files, do one of the following:
 - From the Debugger, select **Browse → Files**.
 - From the File Locator located on the Debugger navigation bar, select **Browse all source files in program**.
 - From an existing Browse window, select **Object → Files**.
 - From the command pane, issue the **browse files** command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.
- To open a Browse window displaying a more limited selection of source files, issue the **e** command with a suitable pattern (e ***.c**, for example). For information about the **e** command, see Chapter 12, “Navigation Command Reference” in the *MULTI: Debugging Command Reference* book.
- To open a Browse window displaying all of the files included by the current file, right-click an empty spot in the Debugger's source pane and select **Browse Includes Of This File**.

- To open a Browse window displaying all of the files that this file includes, right-click an empty spot in the Debugger's source pane and select **Browse Files Included By This File**.

Browsing Source Files General Information

The name of each source file is listed in the Browse window. Source files are color-coded according to MULTI's color settings. They are displayed in the following way:

- Source files that do not define any procedures are displayed in gray.
- Source file headings are displayed in the color used for keywords (see "Headings in the Source File Browse Window" on page 213).
- All other source files are displayed in the normal text color.

Browsing Source Files Show Menu

The following table describes the options available in the **Show** menu when you browse source files. It also lists which options are enabled by default.



Note The primary column when you browse source files is the **Source File Name** column. The formatting options listed below apply only to this column, which is always displayed when you browse source files.

Option	Default	Meaning
Full Name	Off	Formats the Source File Name column (the primary column) so that the full path is displayed.
Base Name	On	Formats the Source File Name column (the primary column) so that only the actual file name is displayed.
Module	Off	Shows or hides the Module column. This column displays the name of the module in which the source file is located, if any.

Browsing Source Files Mouse Operations

The following table describes the results of mouse operations on source files in the Browse window.

Mouse action	Meaning
Click	Displays the selected source file in the Debugger's source pane.
Double-click	Opens a Browse window of procedures defined in the selected source file, if any.
Right-click	Opens a shortcut menu. For more details, see "Browsing Source Files Tools Menu" on page 213.

Browsing Source Files Tools Menu

The following table describes the options available in the **Tools** menu when you browse source files. The same menu is displayed when you right-click content in the Browse window.

Option	Meaning
Browse Procedures in File	Opens a Browse window of procedures defined in the selected source file, if any.
Contract All	Contracts every heading.
Expand All	Expands every heading (this is the way the Browse window looks when first opened).
Show in Editor	Opens the selected file in MULTI's Editor.
Show in Tree Browser	Opens a Tree Browser window to show the reference relationships between the selected source file and other source files.
Graph Includes	Opens a Graph View window that displays an include file dependency graph centered on the selected source file.

Headings in the Source File Browse Window

As noted in "Browse Window Headings" on page 200, some entries may be headings when you browse source files. For source files, these headings are the directory path of the source files. For example, if there was a Browse window

open on a single file `file.c` contained within the directory `directory`, there would be a single heading labeled `directory` with `file.c` as its only child.

Browsing Data Types

To open a Browse window for data types, do one of the following:

- From the Debugger, select **Browse** → **All Types**.
- From an existing Browse window, select **Object** → **Types**.
- From the command pane, issue the **browse types** command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

Browsing Data Types Show Menu

The following table describes the options available in the **Show** menu when you browse data types. It also lists which options are enabled by default.



Note Only one column is displayed when you browse data types. You cannot hide this column.

Option	Default	Meaning
Name	Off	Formats the displayed names so that the use name is displayed.
Mangled Name	Off	Formats the displayed names so that the mangled name is displayed. For some languages, this may be the same as what would be displayed under the Name option. For example, in C, the mangled name may be the same as the use name.
Unqualified Name	On	Formats the displayed names so that the unscoped portion of the use name is displayed. For example, in C++, a data type <code>bar</code> inside the namespace <code>Foo</code> would have the use name <code>Foo::bar</code> . Under this option, this name would be displayed simply as <code>bar</code> , with a parent heading named <code>Foo</code> . For many entries, this is the same as what would be displayed under the Name option. For more information, see “Headings for Data Types Browse Window” on page 216.

Browsing Data Types Mouse Operations

The following table describes the results of mouse operations on data types in the Browse window.

Mouse action	Meaning
Click	Shows the clicked data type's definition in the Debugger source pane, if it can be found.
Double-click	Opens a new window displaying the cross references for the selected type.
Right-click	Opens a shortcut menu. For more details, see "Browsing Data Types Tools Menu" on page 215.

Browsing Data Types Tools Menu

The following table lists the options available in the **Tools** menu when you browse data types. Additionally, the table indicates which options appear in the shortcut menu that is displayed when you right-click content in the Browse window.

Option	Location	Meaning
View Struct	Both	Opens a Data Explorer to show the selected data type's structure.
Contract All	Both	Contracts every heading.
Expand All	Both	Expands every heading (this is the way the Browse window looks when first opened).
Global Scope On/Off	Tools menu	Toggles whether global scope is enabled or not. See "Headings for Data Types Browse Window" on page 216.
Browse References	Both	Shows the clicked data type's cross references in a new Browse window.
Browse Superclasses	Both	Shows the clicked data type's superclasses (if any) in a new Browse window.
Browse Subclasses	Both	Shows the clicked data type's subclasses (if any) in a new Browse window.

Option	Location	Meaning
Show in Editor	Both	Shows the clicked data type's definition in an Editor Window.
Show in Tree Browser	Both	Shows the clicked data type in a Tree Browser window so that you can browse its hierarchy information.

Headings for Data Types Browse Window

As noted in “Browse Window Headings” on page 200, some entries may be headings when you browse data types. For data types, these headings are the “scopes” (or enclosing types) of the data types listed as children of the heading. For example, in C++ we might have a namespace `Foo` with a member class `Bar`. The full use name of this type would then be `Foo::Bar`, with `Foo` being the scope and `Bar` the unqualified name of the type.

You can collect all the C-style structs in your program into a global scope by using **Tools** → **Global Scope On/Off** to enable the global scope option (see “Browsing Data Types Tools Menu” on page 215). This places all C-style structs into a fake scope, which allows you to hide all of these types by contracting the heading labeled `<global_scope>`.



Note In C++, we also treat template types as scopes over their parameter lists. For example, displaying the class `Foo<int>` in a Browse window would have `Foo` as a heading and `<int>` as a child beneath it. This is so that multiple instantiations of the same base template type do not clutter the display of data types, and so that they are easy to hide (by contracting the children of a template type you do not want to examine).

Browsing Cross References

When the program being debugged is compiled with cross reference information, you can open a Browse window for cross references by right-clicking a symbol in the Debugger's source pane and selecting **Browse References** from the shortcut menu that appears.

Browsing Cross References General Information

Cross references are color-coded in the Browse window according to MULTI's color settings. They are displayed as follows:

- Definitions are displayed in the normal text color.
- Declarations and common declarations are displayed in the color for dead code.
- Reads are displayed in the color for comments.
- Writes are displayed in the color for strings.
- Reads and writes (that is, cross references that both read and write) are displayed in the color for characters.
- Address references are displayed in the color for keywords.

Browsing Cross References Show Menu

The following table describes the options available in the **Show** menu when you browse cross references. It also lists which options are enabled by default.



Note The primary column when you browse cross references is the **File Position** column. This column is always displayed when you browse cross references.

Option	Default	Meaning
Breakpoint	On	Shows or hides the BP column. This column displays a breakdot if the reference lies on a line with associated code. If there is a breakpoint set at that line already, this column displays the appropriate breakpoint icon. To set or clear a breakpoint on this line, click the green dot.
Position in Proc	On	Shows or hides the Procedure Position column. This column displays the procedure-relative line position of the reference. If the cross reference is not located in a procedure, it has no procedure-relative line position and this column is empty.
Module	Off	Shows or hides the Module column. This column displays the name of the module in which the reference is located, if any.

Option	Default	Meaning
Column	Off	Shows or hides the Column column. This column displays the column position of the reference.
Type	On	Shows or hides the Type column. This column displays one of the following values: • <code>Def</code> if the cross reference is a definition. • <code>Decl</code> if the cross reference is declaration. • <code>C-Decl</code> if the cross reference is a common declaration. • <code>Write</code> if the cross reference is an assignment or write to the item. • <code>Read</code> if the cross reference is an access or read from the item. • <code>Read,Write</code> if the cross reference both reads from and writes to the item. For example, in C, <code>item++</code> both adds one to <code>item</code> (a write) and returns the original value (a read). • <code>Addr</code> if the cross reference is an address reference (that is, taking the address of the item).

Browsing Cross References Mouse Operations

The following table describes the results of mouse operations on cross references in the Browse window.

Mouse action	Meaning
Click	Displays the selected cross reference in the Debugger's source pane, and highlights the piece of code for the cross reference. If you click in the BP column on a green dot, the Debugger inserts a breakpoint at the location. If you click in the BP column on a breakpoint icon, the Debugger removes the breakpoint.
Double-click	Opens the selected cross reference in the Editor.
Right-click	Opens a shortcut menu. For more details, see "Browsing Cross References Tools Menu" on page 219.

Browsing Cross References Tools Menu

The following table describes the options available in the **Tools** menu when you browse cross references. Additionally, the table describes shortcut menu options that become available when you right-click content in the Browse window. The table's "Location" column specifies whether the option appears in the **Tools** menu, the shortcut menu, or both menus.

Option	Location	Meaning
Show in Editor	Both	Opens the selected cross reference in the Editor.
Breakpoint and tracepoint menu items	Shortcut menu (BP column only)	For descriptions of these menu items, see "Browsing Procedures Tools Menu" on page 205.
Set BP on Entries	Both	Sets breakpoints at all lines for all displayed cross references.
Delete BP on Entries	Both	Deletes breakpoints at all lines for all displayed cross references.
Breakpoint Window	Shortcut menu (BP column only)	Opens a Breakpoints window for the program being debugged.

To set breakpoints at all of the locations where the value of a variable (or a type's member field) is changed:

1. Obtain the references for the variable. (In the Debugger source pane, right-click the variable, and select **Browse References**.)
2. In the Browse window, hide the right-value references (select **Filter** → **Hide Right-values**).
3. Right-click anywhere in the Browse window's source pane, and select **Set BP on Entries** from the shortcut menu.

The Tree Browser Window

You can use the Tree Browser to examine the structure of your program in several ways.

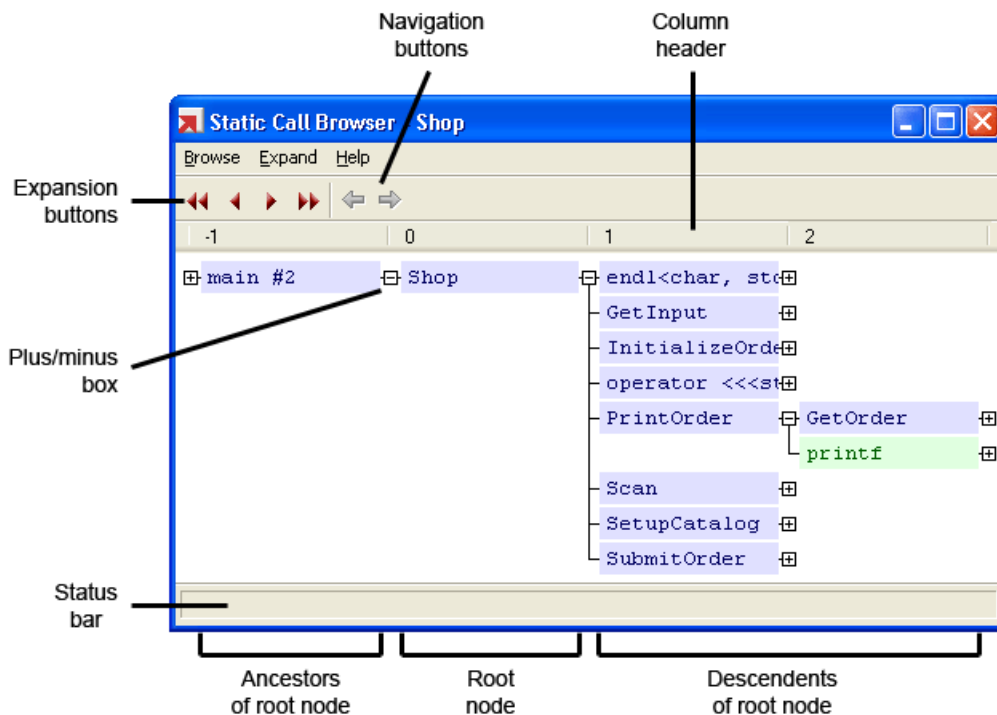
Opening a Tree Browser

To use a Tree Browser, you must be debugging a program. You can open a Tree Browser in several ways, depending on what type of information you want to view. See also:

- “Browsing Classes” on page 224
- “Browsing Static Calls By Function” on page 225
- “Browsing Static Calls By File” on page 226
- “Browsing Dynamic Calls by Function” on page 226

Using a Tree Browser

Regardless of the type of information you are viewing in the Tree Browser, the interface is basically the same.



The main part of the Tree Browser window is a tree graph, which consists of colored nodes. The meanings of the colors vary depending on the type of information you are viewing. (The following sections provide further explanation of coloring.) To customize the colors used to display information in a Tree Browser, see the `TBtypeFG` and `TBtypeBG` configuration options in “Other Debugger Configuration Options” in Chapter 9, “Configuration Options” in the *MULTI: Editing Files and Configuring the IDE* book.

One node of the tree graph is called the “root node.” It is the particular function (or other object) that you are examining. The name of the root node is displayed in the title bar of the Tree Browser window. You can expand ancestors—callers, if you are looking at a function, or superclasses, if you are looking at a class—of the root node towards the left, and descendents—callees or subclasses—of the root node towards the right. You may expand away from the root node for as many levels as you want. However, if you want to look at an ancestor of a descendent of the root node, for example, you will need to reroot your graph. See “Rerooting” on page 223.

To expand ancestors or descendents of a node, click the plus sign (⊕) next to the node. To contract something you have expanded, click the minus sign (⊖). If there is no plus or minus sign, there is nothing to expand or contract.

There are four ways to expand many nodes at once. They are available from the **Expand** menu or from the expansion buttons on the toolbar.

To expand the descendents of the selected node (or of the root node if no node is selected) until there are no more descendents, until recursion is detected, until 50 levels have been expanded, or until 10,000 new nodes have been revealed by expansion, do one of the following:

- Select **Expand → All Descendents**.
- Click **Expand All Descendents** (▶▶).



Note To cancel this operation, press Esc.

To perform a similar expansion on the ancestors of the selected node (or on the ancestors of the root node if no node is selected), do one of the following:

- Select **Expand → All Ancestors**.
- Click **Expand All Ancestors** (◀◀).

Sometimes you may want to expand one level instead of all the nodes. To expand one level of the selected node's descendents (or of the root node's descendents if no node is selected), do one of the following:

- Select **Expand → One Level of Descendents**.
- Click **Expand Descendents One Level** ()

To obtain similar results, you can also click every node's plus sign on the descendent side of the graph. The difference between doing this and performing one of the preceding operations is that the menu items do not expand recursive nodes.

To expand one level of the selected node's ancestors (or of the root node's ancestors if no node is selected), do one of the following:

- Select **Expand → One Level of Ancestors**.
- Click **Expand Ancestors One Level** ()

To resize a column of nodes, drag the separator on the column header to the right of the column that you want to resize.

Node Operations

Each node is labeled with a short, descriptive name. In C++, the short name does not include class or namespace qualifiers, which come before the final double colon (: :). To view the full name in a tooltip, hover your cursor over the node.

To view more information about a node, click it. Information about the node, including its full name, is displayed in the status bar of the Tree Browser window. Clicking certain types of nodes, such as function and file nodes, also causes the source code for the node to be displayed in the Debugger source pane.

To open a shortcut menu for a node, right-click the node. The shortcut menu for a function node resembles the following:



A class node shortcut menu resembles the following:

Reroot
Reroot in New Window
Browse Members in Class
Examine Class in Debugger
superclasses
subclasses

The first two operations, **Reroot** and **Reroot in New Window**, are described in “Rerooting” on page 223. The middle portion of the menu contains various actions you can perform on the node. These actions vary depending on the type of node and are documented in “Opening a Tree Browser” on page 220. The bottom portion of the menu lists the types of ancestors or descendents a node may have. It also allows you to expand or contract them, which is equivalent to clicking the plus/minus sign on the side of the node.

Rerooting

If there is a node on the graph that you want to make the root node so that you can examine both its ancestors and its descendents, you can reroot on that node.

To reroot on a node, do one of the following:

- Click the node and select **Browse → Reroot Selected Node**.
- Right-click the node and select **Reroot**.
- Middle-click the node.

Once you reroot on a node, everything that was previously in the Tree Browser window disappears. However, the previous content of the window is stored in a history mechanism much like that in a Web browser.

To access the history, do one of the following:

- Select **Browse → Back**, or click **Back** (.
- Select **Browse → Forward**, or click **Forward** (.

To reroot a node in a new window, rather than replacing the current window contents, do one of the following:

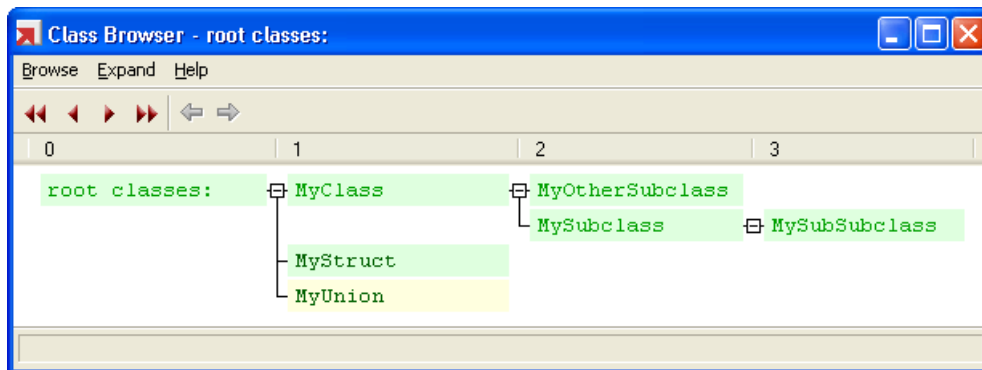
- Click the node and select **Browse → Reroot Selected Node(s) in New Window(s)**.
- Right-click the node and select **Reroot in New Window**.
- Double-middle-click the node.

Browsing Classes

To use a Tree Browser to browse your class hierarchy, do one of the following:

- From the Debugger, select **Browse → Classes**.
- In the Debugger command pane, enter the **browse classes** command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

A window similar to the following appears.



The children of the `root classes` node are all of the classes (including structs and unions) that do not inherit from another class. A class that is a subclass of another class is shown as a child of its parent class. The color green is used to distinguish classes and structs from unions, which are highlighted in yellow.

To view the members in a class, do one of the following:

- Double-click the class.
- Right-click the class and select **Browse Members in Class**.

To see the definition of the class in the Debugger window, do one of the following:

- Click the class.
- Right-click the class and select **Examine Class in Debugger**.

Browsing Static Calls By Function

The Tree Browser can use information from your program's symbol table to show you which functions your functions call, or those they are called by. These potential, or *static*, paths are solely based on the build-time symbol table of your program, and not on the actual run-time paths taken by the process.

To browse static calls by function, do one of the following:

- From the Debugger, select **Browse → Static Calls**.
- In the Debugger command pane, enter the **browse scalls** command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

A Tree Browser opens, centered on the function you are currently looking at in the Debugger source window.

To open a Tree Browser on a specific function (for example, `foo()`), do one of the following:

- In the Debugger, right-click the function and select **Browse Other → Browse Static Calls**.
- In the Debugger command pane, enter the following:

```
browse scalls foo
```

Color provides information about the function represented by a given node. Purple indicates a normal function for which MULTI has the source code. Green indicates a function for which no information is available—usually because the function is in a library that MULTI does not have symbols for. Pink indicates a recursion in the Tree Browser.

To view a function in the Debugger source pane, click the function node.

To edit a function, double-click the function node. This feature is also available from the shortcut menu that appears when you right-click the function node.

Browsing Static Calls By File

In addition to viewing the static call graph by function, you can also view it by file. The root file is connected to any file that contains a function called from within the root file. This may result in the root file being connected to itself.

To browse static calls by file, do one of the following:

- From the Debugger, select **Browse → File Calls**.
- In the Debugger command pane, enter the **browse fcalls** command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

A Tree Browser opens on the file you are currently viewing in the Debugger.

To open a Tree Browser on a specific file (for example, **foo.c**), enter the following in the Debugger command pane:

```
browse fcalls foo.c
```

Color provides information about the file represented by a given node. Pink indicates a file that has debug information. Gray indicates a file that does not have complete debug information.

To view a file in the Debugger, click its node. To edit a file, double-click its node. Right-click a file node to open a shortcut menu that allows you to edit the file, view the file in the Debugger source pane, browse a list of functions in the file, or view file properties.

Browsing Dynamic Calls by Function

Unlike the static call graph, which shows potential calls, the dynamic call graph uses profiling data to display which functions a function actually called during run time. This feature is only available if you have collected call count profiling data with call graph support enabled. See “Overview of Profiling Methods” on page 341.

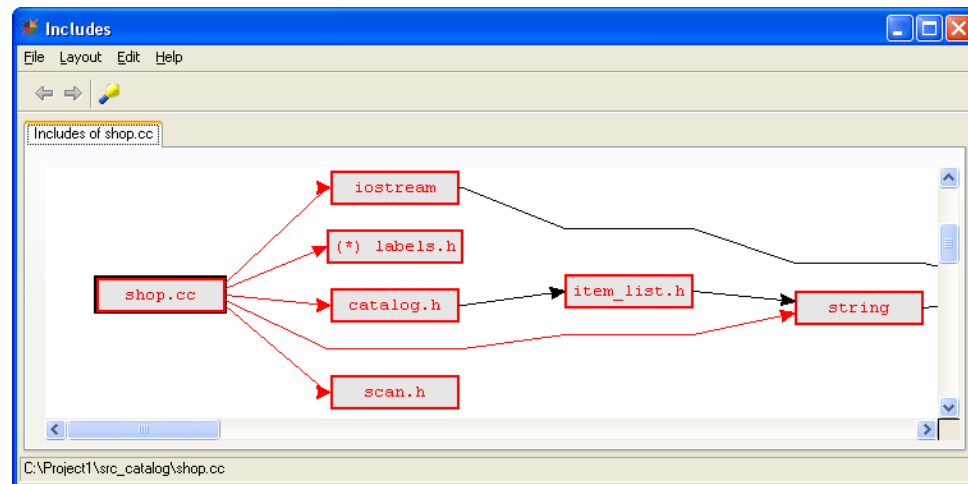
To browse the dynamic call graph, do one of the following:

- From the Debugger, select **Browse → Dynamic Calls**.
- In the Debugger command pane, enter the **browse dcalls** command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book..
- To view a particular function specified by *function_name*, enter **browse dcalls function_name** in the Debugger command pane. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

Pink nodes indicate functions that have debug information. Gray nodes indicate functions that do not have debug information.

The Graph View Window

The Graph View window displays relationships between items as a two-dimensional graph of objects. An edge between two objects in the graph indicates a relationship between the two items.

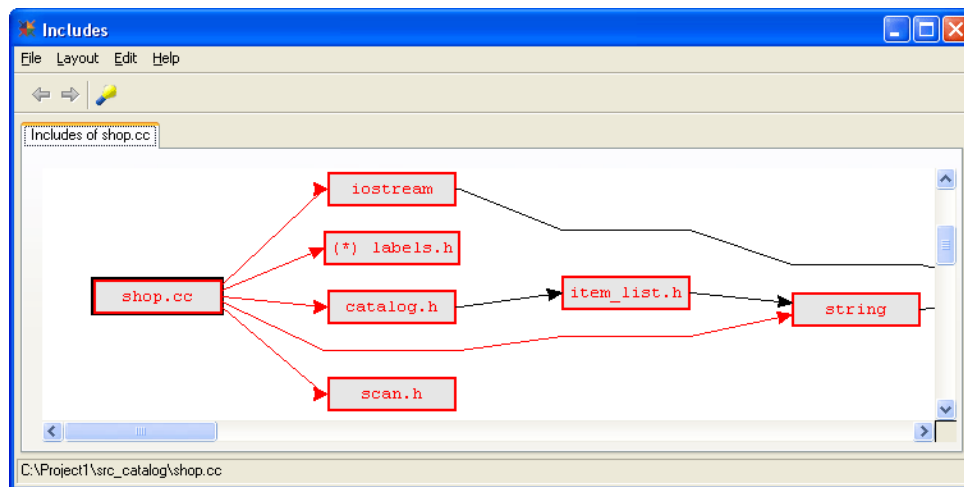


Browsing Includes

To open a Graph View window displaying an include file dependency graph, do one of the following:

- From the Debugger, select **Browse** → **Includes**.
- From the command pane, issue the **browse includes** command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

The resulting graph is centered on the current file, and shows all files included in this file as well as all files that include this file. An edge pointing from a file to an included file indicates a dependency.



When you display the include dependency graph from a root file (that is, a file that is not itself included by anything), some files may be marked with a leading (*). These are files that appear not to contribute anything to the root file and that can probably be removed from the include graph without harm.

Controlling Layout and Navigating in Graph View

You can control the layout of a graph in the Graph View window via the **Layout** menu or the right-click shortcut menu. You can navigate the graph via the right-click shortcut menu, the **Search for Objects** window, or the history buttons. The following sections describe each in more detail.

The Layout Menu

The Graph View **Layout** menu contains the following options:

Option	Meaning
Redo Layout	Recalculates the layout of the graph. This option may be used to improve the layout of the graph after collapsing nodes.
Vertical Layout	Formats the graph vertically (edges are directed downward). When disabled, the graph is formatted horizontally (edges are directed rightward).

The Graph View Shortcut Menu

Right-clicking an object in the graph gives a shortcut menu with the following options:

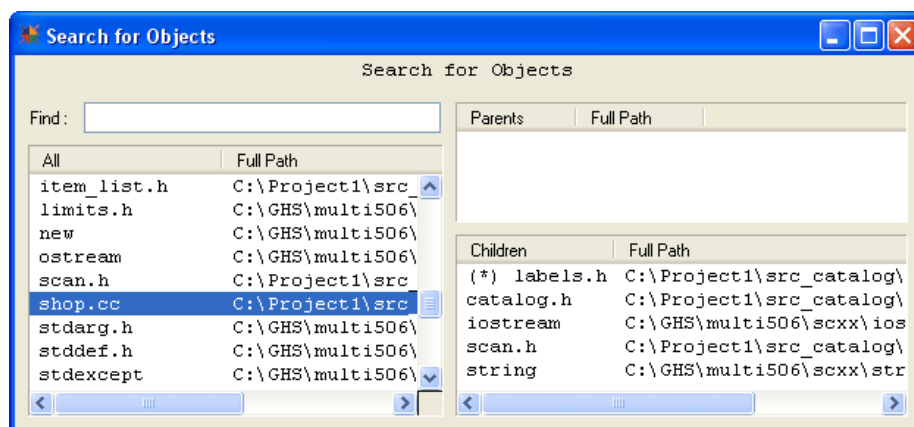
Option	Meaning
Redo Layout	Recalculates the layout of the graph. This option may be used to improve the layout of the graph after collapsing nodes.
Go To Child	Opens a submenu listing children of the selected object. Choosing one of these children scrolls the graph to that object. If the object has more than 10 children, an additional option View Full List appears in the submenu. Selecting this option opens the Search for Objects window.
Go To Parent	Opens a submenu listing parents of the selected object. Choosing one of these parents scrolls the graph to that object. If the object has more than 10 parents, an additional option View Full List appears in the submenu. Selecting this option opens the Search for Objects window.
Collapse Node	Removes the object and all descendants that have only one parent. You can display the object again by selecting Expand Children from the object's parent or by selecting Expand Parent from any of the children that are still visible.

Option	Meaning
Collapse Children	Removes all children of the object and then traverses the subtrees of each child, removing those objects that no longer show parents. In a simple tree, the net effect is to remove all nodes in the subtree rooted at the object. You can display the children again by choosing Expand Children .
Expand Children	Redisplays the object's children, if they were previously collapsed. Also redisplay the children's children, and so on, since they now have a visible parent.
Collapse Parents	Similar to Collapse Children , except operating on the object's parents, the parents' parents, and so on.
Expand Parents	Similar to Expand Children , except operating on the object's parents, the parents' parents, and so on.
Edit File	Opens the file in an editor.
Re-center Graph	Re-centers the graph on the selected file, producing a graph of files that include the selected file, and files that it includes. This creates a new entry in the history.

The Search for Objects Window

The **Search for Objects** window provides additional navigation tools for Graph View. To open the **Search for Objects** window, do one of the following:

- In the Graph View toolbar, click the **Search** button (.
- Select **Edit** → **Search**.



The **Search for Objects** window consists of the following elements:

- The **Find** text field, which allows you to specify a filter for the entries in the **All** list. Filters may include wildcards and are case-insensitive.
- The **All** list displays a list of all objects that fit the current filter. Selecting an item in the **All** list scrolls to and selects the corresponding object in Graph View. Similarly, selecting an object in Graph View selects the corresponding list item in the **All** list.
- The **Parents** list displays a list of all parents of the selected object. Entries that appear dimmed indicate objects that are currently collapsed. Selecting an item in the **Parents** list causes Graph View to scroll to, but not select, the corresponding object.
- The **Children** list displays a list of all children of the selected object. Entries that appear dimmed indicate objects that are currently collapsed. Selecting an item in the **Children** list causes Graph View to scroll to, but not select, the corresponding object.

The History Buttons

The Graph View history buttons allow you to switch between different versions of the graph. Clicking the **Back** button () moves back in the history, while clicking the **Forward** button () moves forward in the history.

Using the Register Explorer

Chapter 11

This Chapter Contains:

- The Register View Window
- The Register Information Window
- The Modify Register Definition Dialog
- The Register Setup Dialog
- The Register Search Window
- Using Debugger Commands to Work with Registers
- Customizing Registers

The Register Explorer allows you to configure how registers are defined, accessed, and displayed within the Debugger. The Register Explorer includes the **Register View** window, which allows you to view the values of registers within the Debugger and the **Register Information** window, which shows you detailed information about a register and allows you to add and change register definitions.



Note Despite its name, the Register Explorer is not limited to working with the physical registers on your target processor. The Register Explorer can also work with objects that derive their values from other registers, from locations in memory, or even from the result of a Debugger expression. All of these objects are treated in the same manner as physical target registers. In this chapter, the term *register* includes everything that can be described in a Register Explorer definitions file.

To use the Register Explorer, your MULTI installation must include one or more register definition files for your target architecture. Most target architectures already have the necessary register definition files. For more information, see “Register Explorer Startup” on page 266. The remainder of this chapter assumes that the Register Explorer is available.

Register definition files are defined in a file format called GRD. For detailed information about the GRD format for defining registers, see “The GRD Register Definition Format” on page 610. The legacy RDF format is no longer supported; it has been superseded by the GRD format.

The Register View Window

The Register Explorer provides a **Register View** window specifically for viewing registers. You can use this window to create multiple views of a target’s registers and easily switch between them. Since the window can hide registers and groups, you can use it to pare down dozens of registers into the ones that are most relevant for debugging your application.

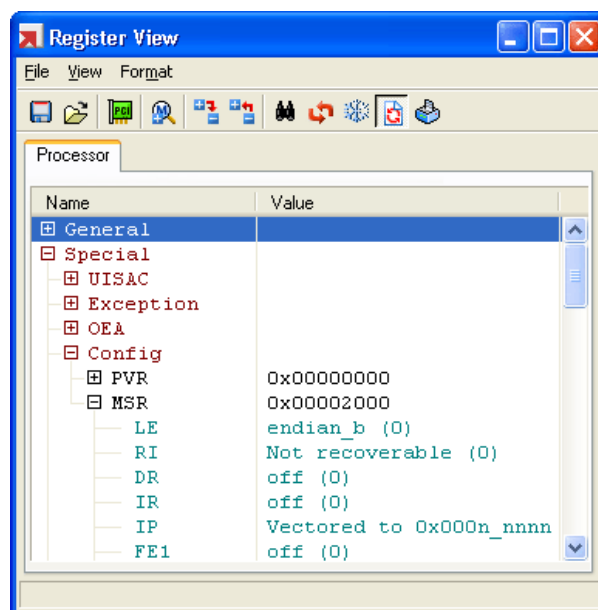
To open the **Register View** window from a Debugger, you can do one of the following:

- Click the **Registers** button () on the Debugger toolbar.
- Select **View** → **Registers**.

- Run the command **regview** in the Debugger command pane. For information about the **regview** command, see “Register Commands” in Chapter 7, “Configuration Command Reference” in the *MULTI: Debugging Command Reference* book.

If the Register Explorer is not active, a regular Data Explorer listing all of your target’s registers will open. For more information about the Data Explorer, see “The Data Explorer Window” on page 165.

The **Register View** window shows all the registers for your target, your custom registers, and various configuration controls. An example PowerPC **Register View** window is shown below.



The main components of the **Register View** window include:

- *Menu Bar* — Contains entries for various operations that allow you to configure the tabs and appearance of the **Register View** window. For a description of the individual menus and options, see “The Menu Bar” on page 236.
- *Toolbar* — Provides shortcuts for often-used operations. For a description of the individual buttons, see “Toolbar” on page 239.
- *View tabs* — Show the registers of your target organized in different ways. For information about the default tabs, see “Tabs” on page 240. For instructions on how to create custom tabs, see “Customizing the Register View Window” on page 243.

- *Register tree* — Each tab contains a two-column register tree. You can select registers and groups from this tree and perform actions on them using the right-click menu. For details, see “Register Tree” on page 241.

The Menu Bar

The **Register View** window has three menus, **File**, **View**, and **Format**.

The **File** menu contains the menu items listed in the following table.

Item	Meaning
Save All Register Values into File	Saves values of all registers into a text file.
Save All Register Values in Active Tab into File	Saves the values of all registers in the current tab into a file.
Save Selected Register Values into File	Saves the values of the selected registers in the current Tab into a file
Load Register Values from File	Loads register values from a file in the saved register value format. See a saved register value file for an example.
Save Register Definitions Created on the Fly	Saves to a .grd file the register definitions modified during the current debugging session, registers defined with the regadd command, or registers defined from a Data Explorer’s View → View “variable” as Register menu item. See “The Register Setup Dialog” on page 259.
Load Register Definitions from File	Loads register definitions from a .grd file and adds them to the definitions already present.
Unload Register Definitions from File	Unloads register definitions from an incrementally loaded .grd file.
Add Memory-mapped Register	Creates a memory-mapped register for a memory address.
Reinitialize Register Information	Reinitialize the register information from the default register definition file(s). This is a slow process for some targets.
Reuse Register Information Window	Toggles the option that specifies whether to reuse an existing Register Information window or open an additional Register Information window when a new register is viewed.

Item	Meaning
Freeze	Freezes the Register View window so that it does not automatically refresh. When the Register View window is not frozen, the values of the registers in the window are updated each time the process stops. If the window is frozen, the register values will not be updated each time the process stops.
Print	Prints the register values.
Close	Closes the Register View window.

The **View** menu contains the following menu items.

Item	Meaning
Search Register	Launches the Register Search window, see “The Register Search Window” on page 260).
Show Hidden Registers	Controls whether registers that are hidden on the active tab are displayed in the register tree. When a check mark appears beside this menu item, registers and groups that have been hidden on the active tab will be displayed in the register tree as light gray entries. When hidden items are displayed, you can edit them to remove the hidden attribute. When no check mark appears next to this menu item, registers and groups that are hidden on the active tab are not displayed in the register tree. To toggle between these settings, select the Show Hidden Registers menu item.
Add New Tab	Adds a new tab to the Register View window. See “Adding a New Register View Window Tab” on page 244.
Remove Active Tab	Deletes the active tab from the Register View window. See “Removing Unwanted Tabs” on page 246.
Promote Active Tab	Moves the tab one to the left in the ordering of tabs at the top of the Register View window. See “Modifying the Ordering of Tabs” on page 245.
Demote Active Tab	Moves the tab one to the right in the ordering of tabs at the top of the Register View window. See “Modifying the Ordering of Tabs” on page 245.
Unroot Active Tab	Makes the active tab display groups starting from the root of the register group tree, if the tree has been re-rooted. See “Changing the Root of the Register Tree” on page 246.

Item	Meaning
Refresh	Refreshes all of the register values displayed in the register tree if the Register View window is not frozen.
Expand All	Expands all nodes on the current tab of the Register View window.
Collapse All	Contracts all nodes on the current tab of the Register View window.

The **Format** menu contains the menu items listed in the table below. You can use this menu to configure the tabs and appearance of the **Register View** window.

Item	Meaning
Natural	When selected, register values will be displayed in their default format. This default format corresponds to how values of the register's type are normally displayed in Data Explorers.
Decimal	When selected, register values will be displayed in signed base 10 integer notation whenever applicable.
Hexadecimal	When selected, register values will be displayed in unsigned base 16 notation whenever applicable.
Binary	When selected, register values will be displayed in unsigned binary notation whenever applicable.
Octal	When selected, register values will be displayed in unsigned base 8 notation whenever applicable.
View Alternate	When selected, register values will be displayed in the natural format as well as in a secondary format. For example, an enumeration value is displayed along with an integer value when this menu item is selected.
Pad Hex Values	When selected, any hexadecimal values that are displayed will include leading zeros to their full bit width. When cleared, hexadecimal values are displayed without leading zeros.
Expand Value	When selected, any registers that are pointers to values stored in memory will be automatically dereferenced and the contents of memory at that location will be displayed. When cleared, only the pointer is shown and not the value to which it points.

Item	Meaning
Show Changes	When selected, any register values that change while you are stepping through your program are highlighted to make them distinct from registers whose values did not change. When cleared, all registers are displayed in the same way, regardless of whether their values changed or not.
Expand Register Bitfields	When selected, the bitfields contained within registers will be displayed as structures (individual values separated by vertical bars). When cleared, the bitfield registers are displayed as integer hexadecimal or decimal constants.

Toolbar

Directly underneath the menu bar is the toolbar, which contains the following buttons:

Button	Meaning
	Saves the values of the selected registers in the current Tab into a file.
	Loads register values from a file in the saved register value format. See a saved register value file for an example.
	Creates a memory-mapped register for a memory address.
	Expands all nodes on the current tab of the Register View window.
	Contracts all nodes on the current tab of the Register View window.
	Launches the Register Search window (see “The Register Search Window” on page 260) so that you can search a register you want.
	Refreshes all of the register values displayed in the register tree if the Register View window is not frozen.
	Freezes the Register View window so that it does not automatically refresh. When the Register View window is not frozen, the values of the registers in the window are updated each time the process stops. If the window is frozen, the register values will not be updated each time the process stops.

Button	Meaning
	Toggles the option that specifies whether to reuse an existing Register Information window or open an additional Register Information window when a new register is viewed.
	Prints the register values.
	Closes the Register View window. Whether this button appears on your toolbar depends on the setting of the option Display close (x) buttons . To access this option, select Config → Options → General Tab .

Tabs

Directly underneath the toolbar is a list of tabs. Each tab of the **Register View** window shows the registers of your target organized in a different way. In addition to tabs that you create, you may see default tabs, which are automatically created. The possible default tabs are listed below:

- The **Processor** tab usually contains all of the registers present on the target processor.
- The **Board** tab contains all memory-mapped registers and dynamic registers defined in the Debugger, as well as indirect registers. These registers are usually specific to the target board you are using.
- The **Ungrouped** tab displays any user-defined registers that have not been placed into any register group.

These default tabs are present only if they contain registers. For example, the **Ungrouped** tab will probably not appear unless you have defined your own registers and these registers have not been placed in any group.

Each register tab except for **Ungrouped** views the same hierarchical register structure as defined in the register definition files. Different tabs can be configured to display different subsets of the available registers. You can alter the visibility of registers and groups on a single tab without affecting other tabs. See “Selectively Displaying Registers and Groups” on page 244.

To switch between tabs, click the name of the tab you want to view in the tab list. For information about adding a new tab or performing other operations on tabs, see “Customizing the Register View Window” on page 243.

Register Tree

Underneath the tab list is a tree of the registers and groups that are visible for the active tab. The column on the left displays the names of register groups and the names of registers. The column on the right shows the current value for registers. If the list of registers and groups is larger than the visible area of the register tree, scroll to view parts of the tree that are not visible.

Each group in the left column has a plus or minus icon next to it. To expand or contract the group, click this icon. When the group is expanded, the minus sign icon will be shown and all of the items contained in that register group will be visible in the register tree. When the group is contracted, the plus sign icon will be shown and the items contained in that register group will not be visible in the register tree. Registers containing a pointer also display plus/minus icons.

To select a single row of the register tree, click the row. To select a contiguous range of rows of the register tree, use Shift + [click]. To make a noncontiguous selection of rows, use Ctrl + [click].

To operate on an entry in the register tree, right-click the entry. A shortcut menu will appear. For more information, see “Performing Actions on Registers Using the Shortcut Menus” on page 242.

To open a Data Explorer (see also “The Data Explorer Window” on page 165) or a **Register Information** window (if the register has bitfields definition) on a register that is in a row of the register tree, double-click the row.

In the **Name** column, a tooltip may be available:

- For registers or groups that have names that extend beyond the width of the **Name** column;
- For registers or groups that have an alternate name specified in the GRD file with the `long_name` or `ln` option;
- For registers that have certain special values, such as a pointer to code.

Colors are used to distinguish different objects:

- Groups are displayed in the color MULTI uses for string constants;
- Registers are displayed in the standard text color;
- Register bitfields are displayed in the color MULTI uses for customized items;

- Dereferences of pointer registers are displayed in the color MULTI uses for keywords.

Performing Actions on Registers Using the Shortcut Menus

When you right-click a row or selection in the register tree, a shortcut menu appears. You can use this menu to change properties of the selected items, as well as the active tab of the **Register View** window. Three types of objects can be right-clicked in the **Register View** window: a single register, a single group, or a selection that contains multiple items of the register tree. A shortcut menu appears that contains some of the items from the following table.

Menu Item	Meaning	Shown For
Hide <i>Item</i> or Hide Selected Registers	If the selection was visible in the register tree of the active tab, this choice will make it invisible (unless Show Hidden Items is selected). When you hide a group, all of its children are implicitly hidden as well. See “Selectively Displaying Registers and Groups” on page 244.	Single register Single group Multiple items
Show <i>Item</i> or Show Selected Registers	If the selection was hidden in the register tree of the active tab, this choice will make the selection visible again. See “Selectively Displaying Registers and Groups” on page 244.	Single register Single group Multiple items
Edit Contents of Register	Opens a dialog box that allows the value of the register to be modified. Editing values through this dialog is only possible for registers that do not contain structures or bitfields. Complex structures must be edited from the command line or a Data Explorer. See “Editing Register Contents” on page 247.	Single register
Open a Data Explorer on Register	Opens a Data Explorer displaying the value of the selected register or its field. See “The Data Explorer Window” on page 165.	Single register

Menu Item	Meaning	Shown For
Open an Info View on Register	Opens a Register Information window displaying the selected register's detail information. See "The Register Information Window" on page 250.	Single register
View Memory at Address in Register	Opens a Memory View window displaying the memory at the address represented by the selected register. See Chapter 13, "Using the Memory View Window".	Single register
Reroot Tab at Group	Makes the tab display only the children of the selected group. This is useful when configuring new tabs. See "Changing the Root of the Register Tree" on page 246.	Single group
Unroot Active Tab	Reverts the effect of any rerooting commands applied to the tab. This is useful when configuring new tabs. See "Changing the Root of the Register Tree" on page 246.	Single register Single group Multiple items
Copy Values	Copies the contents of the value column for the selected item or items to the clipboard. See "Copying Register Values" on page 247.	Single register Single group Multiple items
Save Selected Register Values into File	Saves the values of the selected registers in the current Tab into a file.	Single register Single group Multiple items
Load Register Values from File	Loads register values from a file in the saved register value format. See a saved register value file for an example.	Single register Single group Multiple items

Customizing the Register View Window

In addition to configuring the basic display of the **Register View** window, you can also create and store multiple display configurations that will appear as customized tabs in the **Register View** window.

Each tab in the **Register View** window contains a distinct view of the hierarchical organization of groups and registers. You can choose to display

only certain registers and groups on each tab independently of other tabs. This means that you can create new tabs that only display the registers you are interested in at particular points in your process, and flip between these displays by clicking the appropriate tabs.



Note You cannot modify the way registers are hierarchically grouped or ordered without modifying the register definition files.

The **Register View** window remembers how it was configured between debugging sessions. Settings are saved and restored based upon which register definition file is loaded when the Debugger is opened. This usually means that configurations of the registers for different target processor families are saved and restored independently of each other.

Adding a New Register View Window Tab

The first step in creating a new configuration of the **Register View** window is to add a new tab. To do this, choose **View → Add New Tab** in the **Register View** window. Use the dialog box to enter the name you want to give the new tab. Although not necessary, this name should be short to keep the tab list narrower than the width of your **Register View** window, making it easier to switch tabs without scrolling through a long tab list. After you click **OK**, a new tab will appear in the list at the top of the **Register View** window and will be made the active tab.

When you create a new tab, it displays the full structure of the register tree by default. You can now show or hide registers and groups to pare this tree down to the information that is most important to you.

Selectively Displaying Registers and Groups

Some target processors may have dozens of registers that are displayed in the default register tree. You may not care about the values of many of these registers, and the ones that are important to you may be scattered throughout the register tree. By changing which items of the register tree are shown or hidden, you can display only the information that is most relevant to you. Registers and groups are hidden on a per-tab basis; hiding or showing a group on one tab does not affect its display on the other tabs.

To hide a register, right-click the register you want to hide and choose **Hide Register** from the shortcut menu.

To hide groups of registers, right-click the group and choose **Hide Group** from the shortcut menu. The group, and anything the group contained, will no longer be displayed on that tab.

To hide multiple objects, use Shift + [click] or Ctrl + [click] in the register tree to select multiple items. When you have selected everything you want to hide, right-click one of the selected items and choose **Hide Selected Items** from the shortcut menu.

To display a hidden register or group, choose **View → Show Hidden Registers**. All of the hidden registers and groups are now displayed as light gray items in the register tree. Right-click the item you want to display and choose **Show Item** from the menu. The name of the item now appears in black, indicating that it is no longer hidden. To show multiple items simultaneously, right-click a member of a multiple selection, similarly to how multiple items are hidden at once. To remove the dimmed items from the list, again choose the **View → Show Hidden Registers** option.

You can also show and hide items on a tab by using the **regtab** command. For more information about this command, see “Register Commands” in Chapter 7, “Configuration Command Reference” in the *MULTI: Debugging Command Reference* book.

Modifying the Ordering of Tabs

You can modify the left to right tab order that appears at the top of the **Register View** window. The order of the tabs is maintained across Debugger sessions. Whenever you open a new **Register View** window, the leftmost tab in the ordering is the first one that is displayed.

To move a tab one position to the left, first click its label to make it the active tab. Then choose **View → Promote Active Tab** in the **Register View** window.

To move a tab one position to the right, first click its label to make it the active tab. Then choose **View → Demote Active Tab** in the **Register View** window.

You can also modify the ordering of tabs in the **Register View** window by using the **regtab** command. For more information about this command, see “Register

Commands” in Chapter 7, “Configuration Command Reference” in the *MULTI: Debugging Command Reference* book.

Removing Unwanted Tabs

Each open **Register View** window must contain at least one tab. If the window contains more than one tab, you can perform one of the following actions to remove an unwanted tab:

- Click the tab’s label to make it the active tab, and then select **View → Remove Active Tab**.
- Use the **regtab** command. For information about this command, see “Register Commands” in Chapter 7, “Configuration Command Reference” in the *MULTI: Debugging Command Reference* book.

Removal is permanent for customized tabs (see “Adding a New Register View Window Tab” on page 244 and “Register View Window Configuration Files” on page 249). However, the automatically created **Processor**, **Board**, and **Ungrouped** tabs can be restored. To restore these tabs:

1. Exit MULTI.
2. Remove the relevant **.ini** configuration file from the **regview** subdirectory of your personal configuration directory (see “Default Loading of Configuration Files” on page 635). Note that removing this configuration file also permanently deletes all customized tabs.

Changing the Root of the Register Tree

Newly created tabs always display the full register tree. You may only be interested in the registers in a certain group that is nested within other groups of the register tree. By showing and hiding groups, you can make all the other groups invisible, leaving only the registers you are interested in visible, along with the groups that contain them.

To hide the parents of a visible register or group, change the root of the register tree for a tab. When you change the root of the register tree on a tab, only the children of the group chosen to be the new root are displayed. To specify that a group should be the new root, right-click the group and choose **Reroot Tab at Group** from the shortcut menu. Afterwards, the register tree on that tab now displays only the children of the group you specified to be the root.

You can display all of the groups in the default register tree and undo a rerooting operation by unrooting the register tree on a tab. First click the desired label to make it the active tab. Choose **View → Unroot Active Tab** in the **Register View** window. The tab will now display all of the default register tree's visible groups.

You can also reroot and unroot tabs by using the **regtab** command. For information about this command, see “Register Commands” in Chapter 7, “Configuration Command Reference” in the *MULTI: Debugging Command Reference* book.

Copying Register Values

You can place a copy of any information in the register tree onto the clipboard. To make a copy, select the items you want to copy and press Ctrl+C. All of the rows that are selected in the register tree will be copied to the clipboard. Since only full rows can be selected, the copy will take names of groups and registers as well as their values.

Sometimes you may want to copy just the value of a particular register or set of registers and omit the register names. To do this, first select the registers whose values you want to copy. Then right-click the selection and choose **Copy Values** from the shortcut menu that appears. If you only want to copy the value of a single register, right-click the register and it will be selected automatically.

Editing Register Contents

Many registers contain values that are not interpreted as bitfields or structures, but as simple unsigned bits, floating-point values, or integers. For registers that lack complicated structures, their contents can be edited directly from the **Register View** window.

To edit the contents of a register, right-click the register you want to modify and choose **Edit Contents of Register** from the shortcut menu that appears. A dialog box will appear that shows the current value of the register. Type in the new value or a Debugger command line expression that evaluates to the new value you want the register to have. When you click **OK**, the register will be assigned either the value you typed in or the result of evaluating your expression in the current Debugger environment.

To modify the values of registers that have complicated structures, you must do your editing from a **Register Information** window or a Data Explorer. To modify register value with the **Register Information** window:

1. Open a **Register Information** window on the register (right-click the register you want to modify and choose **Open an Info View on Register** from the shortcut menu). A **Register Information** window will appear showing the details of the register's structure.
2. Edit the register's contents as described in "Changing a Register Value" on page 255.

To modify register value with the Data Explorer:

1. Open a Data Explorer on the register (right-click the register you want to modify and choose **Open a Data Explorer on Register** from the shortcut menu). A Data Explorer will appear showing the details of the register's structure.
2. Edit the register's contents as you would edit the values of any other structure in a Data Explorer.

See Chapter 9, "Viewing and Modifying Variables with the Data Explorer".

Controlling Refresh of Register Values

By default, the **Register View** window tries to update the values of all the registers it displays whenever your process is halted. There may be cases where you do not want the values to be automatically updated, such as when you want to keep a copy of the current values of a processor's registers to compare with the values at a future time. You may also want to suppress automatic updating if you are using a slow target connection.

To suppress the automatic updating of register values, click the  button in the **Register View** window. The button will appear to be pushed down to indicate that the window is frozen. While the window is frozen, the register values will not be automatically updated from your target. In the frozen state you may still scroll through the register tree to view all of the register values that were present when the window was frozen. You are not allowed to switch between tabs when the **Register View** window is frozen. To unfreeze the window and allow the register values to be automatically refreshed again, click the  button again.

By default, when the value of a particular register changes, that register's entry will be highlighted in the register tree. To toggle this highlighting on and off, choose **Format** → **Show Changes**.

If you want to manually refresh register values in the **Register View** window while your process is halted, choose **View** → **Refresh**. Usually the values will be automatically refreshed each time the process is halted, but you will need to manually refresh to see any changes to volatile registers that occur while your process is halted.

Printing the Window Contents

To print the list of registers and register values in a **Register View** window, click the  button or select **File** → **Print** and specify the appropriate settings in the **Print Options** dialog that appears.

Register View Window Configuration Files

The **Register View** window tab configurations are stored in configuration files to maintain settings across Debugger sessions. Usually you will not need to modify these configuration files directly. If you are customizing MULTI for internal distributions, however, you may want to add tabs for your users in addition to the **Processor**, **Board**, and **Ungrouped** tabs. To do this, you can do one of the following:

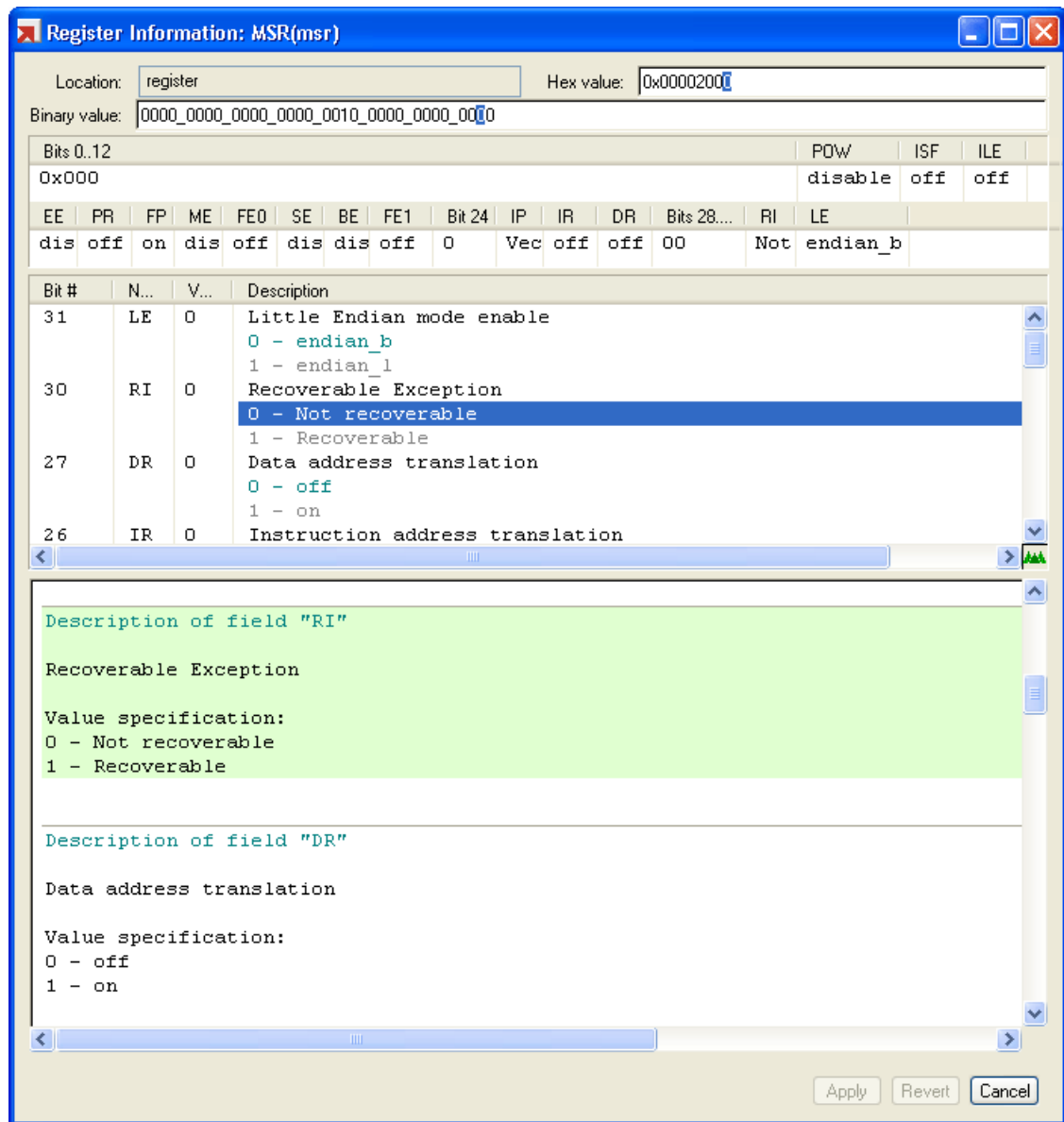
- Use the `gui_tab` clause in the register definition file (for details, see “The GRD Register Definition Format” on page 610) or
- Create a **Register View** window configuration file and install it in the configuration directories of your users. If you want to create such customized configurations, see “Register View Window Configuration Files” on page 634.

The first approach is preferable in most cases.

The Register Information Window

The **Register Information** window shows detailed information about a register, including bitfields and documentation, and provides convenient approaches to change the register's value.

Here is an example for the PowerPC `msr` register:



You can open a **Register Information** window in the following ways:

- In the **Register View** window's shortcut menu, choose **Open an Info View on register_name**.
- In the **Register View** window, double-click a register with a bitfield definition.
- Issue the **regview** command followed by a register name. For information about the **regview** command, see "Register Commands" in Chapter 7, "Configuration Command Reference" in the *MULTI: Debugging Command Reference* book.

The **Register Information** window contains the following parts:

- **Location** — Displays the register's type (register, memory-mapped, etc.) and address information, if any.
- **Hex value** — Displays the register's value, in hexadecimal.

The currently selected hexadecimal digits in the bitfield panes (see below for more information) are displayed in the color used for selections, and any digits changed in this display but not yet written to the target by clicking **Apply** are displayed in red.

- **Binary value** — Displays the register's binary value.

The bits are displayed in groups of 4 or 8 bits. To toggle between 4 and 8 bit groups, select **Show Binary in Nibbles** from the shortcut menu accessible in the bitfield panes. Bit groups are separated by an underline. As in the **Hex value** display, the currently selected binary digits in the bitfield panes (see below for more information) are displayed in the color used for selections, and any bits changed in this display but not yet written to the target by clicking **Apply** are displayed in red.

- Concise display pane — Displays the register's bitfield names as column headings and values in the corresponding cells. Any bits changed in this display but not yet written to the target by clicking **Apply** are displayed in red. For more information, see "Concise Display Pane" on page 252.
- Detailed display pane — Displays the register's detailed bitfield information. Any bits changed in this display but not yet written to the target by clicking **Apply** are displayed in red. For more information, see "Detailed Display Pane" on page 253.

- **Help pane** — Displays the register’s documentation. For details, see “Help Pane” on page 254.

Whenever you select a bitfield in the bitfield panes, its description (if defined in the register definition file) will be shown in help pane in a special background color.

- **Button set** — Allows you to write any changes made in this window to the target, revert any changes, or close the **Register Information** window. For more information, see “Button Set” on page 255.

Concise Display Pane

In the concise display pane, the register’s field names are shown in column headers, and the corresponding field values are shown below the field names. The fields are displayed in order from the most significant bit to the least significant bit.

When you right-click a value cell, a shortcut menu appears with the following items:

Item	Meaning
Set Value: <i>value</i>	Sets the value to the bitfield. This menu item is displayed for a writable register’s enumeration bitfields or 1 or 2 bit bitfields. The current value is dimmed.
Change Value	Allows you to provide a new value for the bitfields. This menu item is displayed for a writable register’s bitfields that are not enumeration types and that are wider than 2 bits.
Modify Register Definition	Launches a Modify Register Definition dialog (see “The Modify Register Definition Dialog” on page 256) to modify the register’s bitfield definitions. You can save the modified register bitfield definition into a file by choosing File → Save Register Definitions Created on the Fly from the Register View window.
Show Numeric Values	Toggles whether to show numeric or literal values for bitfields in the concise display pane.
Pad Undefined Bitfields	Toggles whether to display undefined bitfields.

Item	Meaning
Show Binary in Nibbles	Toggles whether to display binary values grouped by nibbles (4 bits) or bytes (8 bits).
Reverse Bit Numbering	Reverses the numbering for bits. This option only changes the displayed bit indices; it does not change the register's value.
Reverse Byte Order	Swaps the bytes in the register value displayed in the Register Information window. This option changes the register value in the Register Information window. Click the Apply button to write the modification to the target.

Detailed Display Pane

In the detailed display pane, displayed just below the concise display pane, more comprehensive information about each bitfield is present. The bitfields are listed in the order they are defined in the corresponding register definition file.

The register's bitfields and their possible values (for enumeration bitfields) are displayed in rows in the detailed display pane. The following columns are present in the display:

Item	Meaning
Bit #	Bit number or bit number range for a bitfield. The column's value can be changed by toggling the Reverse Bit Numbering option in the shortcut menu.
Name	Bitfield name.
Value	Bitfield's numeric value.
Description	Bitfield's description. If a bitfield has a single-line description, it will be fully displayed in the column; if a bitfield has a multiple-line description, the first line will be displayed in the column followed by . . . ; if a bitfield has no description but has a long name, the long name will be displayed in the column.

When you right-click a cell, a shortcut menu appears with the following items:

Item	Meaning
Show Possible Field Values	Toggles whether to display the possible values for each bitfield of the register. When the flag is on, an enumeration bitfield's possible values are shown below the bitfield. In the row showing a possible bitfield value, the other columns are blank. Double-click one of these bitfield value choices to change the bitfield to that value.
Other items	Other items in the shortcut menu have the same functions as the corresponding items in the shortcut menu of the concise display pane.

Help Pane

The help pane is below the detailed display pane.

At the top, it displays the register's general description in the following order:

- The register's name, aliases etc
- The register's long name (if any)
- The register's description (if any)

Then, for each bitfield, the help pane displays information in the following order:

- The bitfield's name
- The bitfield's long name (if any)
- The bitfield's description (if any)
- The bitfield's possible values (for enumeration types)

The bitfields are shown in the order they are defined in the corresponding register definition file.

Whenever you click a cell in the concise display pane or the detailed display pane, the help pane will automatically scroll to and highlight the corresponding bitfield's documentation.

Button Set

The button set is at the bottom of the **Register Information** window. It contains the following buttons:

Item	Meaning
Apply	Writes the register value displayed in the Register Information window to the target.
Revert	Discards any changes made to the register value and reloads the current register value.
Close	Closes the Register Information window. If any unwritten changes have been made, those changes will be discarded.

Changing a Register Value

The **Register Information** window provides various ways to change a register's value, provided the register is not read-only:

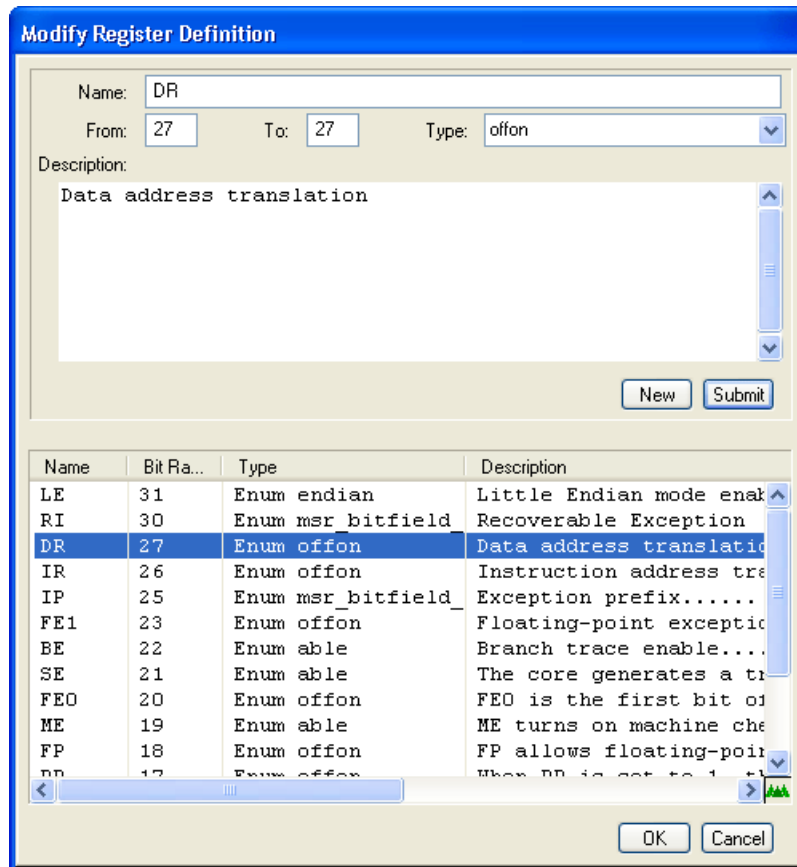
- In the **Hex value**, **Binary value**, the concise display pane's numeric cells and the detailed display pane's **Value** column, you can directly modify the register value:
 1. Select the text you want to change (the hex prefix in **Hex value** cannot be changed);
 2. Type in the new value.

If an invalid character is typed, you may hear a beep.

- In the shortcut menus of the concise display pane and the detailed display pane, you can select the value entries to set the corresponding bitfield's value for enumeration bitfields, or select **Change Value** to type in values for non-enumeration bitfields wider than 2 bits.
- In the detailed display pane, you can double-click a listed enumeration value to set the corresponding bitfield to that value.
- In the concise display pane and the detailed display pane, you can double-click a bitfield with only one bit to toggle its value.

The Modify Register Definition Dialog

You can use the **Modify Register Definition** dialog (an example is shown below) to change a register's bitfield definition for the current debugging session.



The **Modify Register Definition** dialog can be launched by choosing **Modify Register Definition** from the shortcut menu of the **Register Information** window's concise display pane or detailed display pane.

If you modify an existing register's definition or create a new register, you will be asked to save these definitions into a GRD file when you quit MULTI. To save any new or modified register definitions to a GRD file, choose **File** → **Save Register Definitions Created on the Fly** from the **Register View** window. You can merge the changes into your default register definition system later or dynamically load the GRD file in future debugging session by choosing **File** → **Load Register Definitions from File** from the **Register View** window.

The **Modify Register Definition** dialog contains three parts:

- The bitfield editor pane, which contains fields for a bitfield's attributes and a set of buttons to manipulate the changes. For more information, see "Bitfield Editor Pane" on page 257.
- The bitfield list pane, which lists the information for all bitfields. For more information, see "Bitfield List Pane" on page 258.
- The button set, which allows you to commit (**OK**) or discard (**Cancel**) the changes made in the dialog.

Bitfield Editor Pane

The bitfield editor pane has the following attributes. Whenever you select an entry in the bitfield list pane, the settings for the selected bitfield will be displayed in the editor pane. If you click the **New** button, a template for a new bitfield will be displayed in the editor pane.

Item	Meaning
Name	The name of the bitfield. Bitfields within a register must have distinct names. For a new bitfield, MULTI will generate a unique name; you can change it to a more meaningful name.
Description	The description of the bitfield, which will be shown in the Register Information window.
From	The starting bit position (inclusive) of the bitfield. The bits are numbered starting with 0.
To	The ending bit position (inclusive) of the bitfield.
Type	The type of the bitfield: hex, binary, signed, unsigned, or an enumeration type defined in one of the GRD files that have been loaded.

The functions of the buttons in the bitfield editor pane are listed below:

Button	Meaning
New	Creates a new bitfield in the bitfield editor pane.
Submit	Propagates the changes shown in the bitfield editor pane to the bitfield list pane. If the bitfield shown in the bitfield editor pane is an existing bitfield in the bitfield list, the corresponding entry in the bitfield list will be changed; otherwise, the new bitfield will be added to the end of the bitfield list. In order to accept any changes made, you still have to click the OK button in the lower button set.

Bitfield List Pane

The bitfield list pane appears below the bitfield editor pane. Its columns correspond to the attributes of a bitfield, as described in “Bitfield Editor Pane” on page 257.

To display or change a bitfield’s attributes, click a list entry to select it. The bitfield’s attributes will be displayed in the bitfield editor pane, where they can be examined or changed.

The following choices appear in the right-click menu of the bitfield list:

Item	Meaning
Add	Creates a new bitfield in the bitfield editor pane.
Delete	Deletes the selected bitfields.
Modify	Displays the most recently selected bitfield in the bitfield editor pane where it can be examined or changed.
Move Up	Moves the selected bitfield or bitfields up one row.
Move Down	Moves the selected bitfield or bitfields down one row.
Sort by Bit Position in Ascending Order	Sorts the bitfields by their bit position in ascending order.
Sort by Bit Position in Descending Order	Sorts the bitfields by their bit position in descending order.

The following keyboard shortcuts can be used to manipulate the bitfield list:

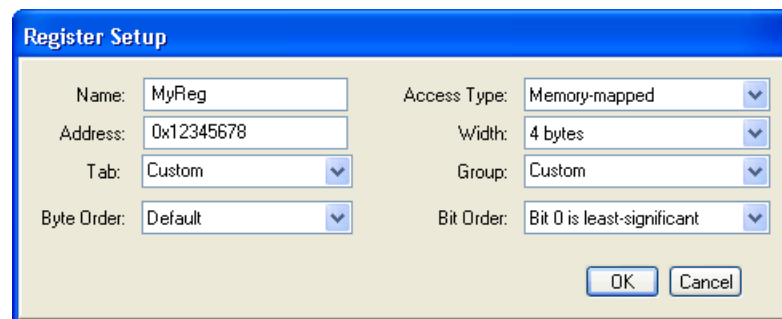
Key Action	Meaning
Ctrl + UpArrow	Moves the selected bitfields up one row.
Ctrl + DownArrow	Moves the selected bitfields down one row.
Ctrl + D	Deletes the selected bitfields.
Delete	Deletes the selected bitfields.

The Register Setup Dialog

You can create a new register in one of these ways:

- Choose **View** → **View “variable” as Register** from a Data Explorer.
- Run the **regadd** command in the Debugger command pane. For information about the **regadd** command, see “Register Commands” in Chapter 7, “Configuration Command Reference” in the *MULTI: Debugging Command Reference* book.

The **Register Setup** dialog opens so that you can specify the basic information for the newly created register.



The **Register Setup** dialog contains the fields listed in the table below.

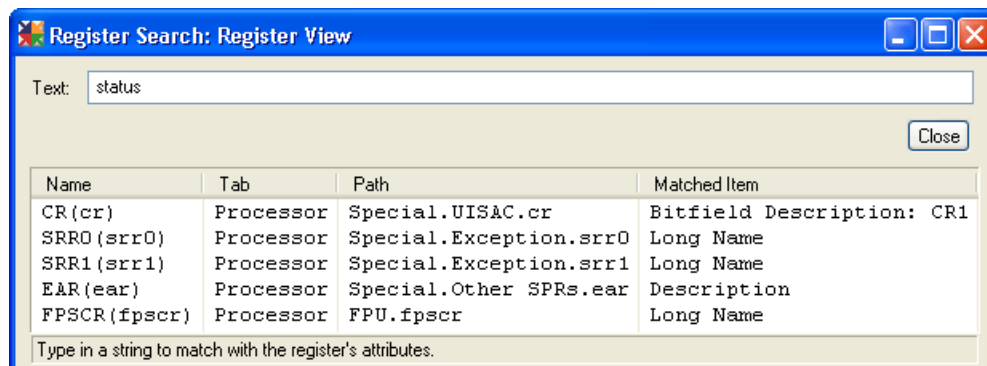
Field	Meaning
Name	Name of the register. This name must not already be used for an existing register.
Access Type	Access type of the register. The only available choice is <i>Memory-mapped</i> .
Address	The address at which the register value can be read.

Field	Meaning
Width	Access size used to read the register value.
Tab	Register View window tab on which the register will be displayed. Choose one of the tab names from the list.
Group	Group to which the register is to be associated. Choose one of the group names from the list.
Byte Order	Specifies the byte order of the register: • Default— Always the same as the target • Big Endian — Always big endian • Little Endian — Always little endian
Bit Order	Chooses whether bits will be numbered with bit 0 referring to the least significant bit, as on most CPU architectures, or with bit 0 referring to the most significant bit, as on PowerPC.

When a new register is created, a new **Register View** window will be launched to show the register in the proper tab, and a **Register Information** window will be launched to show the register's detailed information.

The Register Search Window

To locate a register on a target with many registers, use the **Register Search** window (see an example below).



You can launch the **Register Search** window by either:

- Clicking the  button in the toolbar of the **Register View** window
- Choosing **View** → **Search Register** from the **Register View** window

The **Register Search** window contains three parts:

- **Text** field — Enter a string to be matched against the register's various attributes. After each typed character, MULTI searches the register database loaded from GRD files, and displays matching registers in the Register List. The register attributes searched include:
 - Register names, including name, short name, long name and alias
 - Register description
 - Register bitfield descriptions
- **Close** button — Closes the **Register Search** window.
- Register list — Displays registers matching the text in the **Text** field.

The register list's columns display the information in the following table.

Field	Meaning
Name	Register name.
Tab	Tab in which the register is displayed in the Register View window.
Path	The path of the register in the hierarchy shown in the Register View window. The tab name is at the beginning of the path.
Matched Item	The register attribute that matches the text entered in the Text field.

To locate a register in the **Register View** window, single-click the entry in the register list; to display a register in the **Register Information** window (if the register has bitfields defined) or in the Data Explorer (otherwise), double-click the corresponding entry in the Register List.

The following items appear in the right-click menu of the register list:

Item	Meaning
Show "register" in Register Explorer	Displays the register in the Register View window. The tab to which the register belongs is displayed and all group nodes in the register's path are expanded, if necessary.

Item	Meaning
Show Register Information for “register”	Shows the register in the Register Information window.
Show “register” in Data Explorer	Shows the register in the Data Explorer.

Using Debugger Commands to Work with Registers

Registers that are defined by Register Explorer can be accessed directly from the command line or used in command line expressions. To use the value in the register named *reg* on the command line, enter a dollar sign followed by its name (*\$reg*).

Assume that a 32-bit memory-mapped register *b* is defined and currently contains the value 5. You can display the register value as follows:

```
> $b
$b = 0x00000005
```

You can assign a new value to the register:

```
> $b=3+3
$b = 0x00000006
```

Bitfield registers and structured registers are accessed from the command line in the same way as C-style structs. Assuming that a bitfield register *y* with the following bitfield description in the GRD format is defined as follows:

```
# Bitfields of register y

bitfield {
    y_bitfield {
        "cache_state" {loc="0..1"}
        "reserved" {loc="2..31"}
    }
}
```

This register definition indicates that *y* has two bitfields, *cache_state* in the low two bits and *reserved* in the high 30 bits. Assume the initial value

of `y` is `0x13`. If you print out the register in the command pane, it will be displayed as a C-style struct:

```
> $y
y = struct y {
    reserved = 0x00000001;
    cache_state = 0x00000003;
} y
```

You can display an individual field of a bitfield register. For example:

```
> $y.cache_state
cache_state = 0x00000003
```

You can assign to a field of a bitfield register. For example:

```
> $y.cache_state=2
cache_state = 0x00000002
```

In the above example, the `cache_state` bitfield will be modified without affecting the `reserved` bitfield:

```
> $y
struct y {
    reserved = 0x00000001;
    cache_state = 0x00000002;
} y
```

Registers that are pointers to structures are accessed like C-style pointers on the command line. Assume that a structure `frame` and a register `fp` are described in the register definition files, as follows:

```
# Structure frame definition

struct {
    frame {
        hi_word {type="short"}
        lo_word {type="short"}
    }
}

# some characteristics of register fp

register {
    fp {address=0; type="frame *"}
}
```

Assume that `fp` is currently pointing to a the memory address `0xbbb` that contains `0x000a0007`. The following illustrates what you would see in the command pane when printing out the contents of `fp`.

```
> $fp
$fp : *fp *0xbbb: struct frame {
    hi_word = 10;
    lo_word = 7;
}
```

The pointer was automatically dereferenced when the register contents were printed. To print out a single field, dereference the field:

```
> $fp->lo_word
lo_word = 7
```

To assign to one of the fields, dereference the field:

```
> $fp->hi_word=1 << 3
hi_word = 8
```

Registers that are pointers to raw types are accessed the same way as variables that are pointers to the corresponding type. When using a register's value as an argument to a `MULTI` command or a command line procedure call, use the same syntax as you would for printing out register values or assigning to registers from the command line.

Customizing Registers

Customizing Registers from the Command Line

You can modify existing register definitions and define new registers from the command line while you are debugging a program and the Register Explorer is available. For descriptions of the syntax and function of these commands, see “Register Commands” in Chapter 7, “Configuration Command Reference” in the *MULTI: Debugging Command Reference* book.



Note Any modifications to the register definitions made from the command line are active only until you reload the program or connect to a different target. For persistent modifications, you must use **.rc** files or customize the default register definition files, as described below.

Customizing Registers in Default .rc Files

Any register modifications that are present in a program’s default **.rc** file are persistent across reloads and establishing target connections. Default **.rc** files are the best place to insert definitions of registers you want to consistently use while debugging a specific program. For more information about default **.rc** files, see Chapter 8, “Configuring and Customizing MULTI” in the *MULTI: Editing Files and Configuring the IDE* book.

The **regappend** and **regload** commands can appear in default **.rc** files and will be applied whenever the program is reloaded or a connection to a new target is established. For information about the **regappend** and **regload** commands, see “Register Commands” in Chapter 7, “Configuration Command Reference” in the *MULTI: Debugging Command Reference* book.

Customizing Default Register Definition Files

When you install MULTI, some default register definition files are installed. These files define a set of registers and register groups for popular target processors and boards. You should not need to modify these files. You can customize the default set of files, however, to add new registers or register groups that are relevant to your target environment. This section describes how the default files are located and when they are loaded.



Tip If you are customizing registers for a single program only, you should probably use the default **.rc** file approach. See also “Customizing Registers in Default .rc Files” on page 265. The default register definition files should only be modified if you want definitions to be applied to all of the programs you debug.

Register Explorer Startup

When you open a Debugger, MULTI searches for the root GRD file, which is used to display the registers of the target hardware. If MULTI cannot find or successfully load the root GRD file, the Register Explorer is not active for the debugging session.

The root GRD file is the first of the following four register files that MULTI locates. MULTI searches for the register files in the order listed. For information about the directories MULTI searches when trying to locate the register files, see “File Locations” on page 267.

1. **prog.grd** — *prog* is the name of the program you are debugging. **prog.grd**, if it exists, is a user-defined file.
2. **dbserv-dbtargr.grd** — *dbserv* is the debug server you are connected to, and *dbtargr* is your debug server target. **dbserv-dbtargr.grd**, if it exists, is a user-defined file.
3. **cpu.grd** — *cpu* is the acronym for your processor family. (For a list of possible *cpu* values, see the following table.) **cpu.grd** is included in your MULTI installation. If you have not defined the register files listed above, **cpu.grd** is used as the root GRD file.
4. **dbserv.grd** — *dbserv* is the debug server you are connected to. **dbserv.grd**, if it exists, is a user-defined file.

Appropriate values for *cpu* (in **cpu.grd**) are listed in the following table.

Processor Family	Value of <i>cpu</i> (in cpu.grd)
680x0/683xx	68
Alpha	alp
ARM/Thumb	arm
FirePath	fp
FR	fr20
i960	960

Processor Family	Value of <i>cpu</i> (in <i>cpu.grd</i>)
Lexra	lexra
M-CORE	mc
MIPS/MIPS 16	mips
nCPU	ncp
NDR	ndr
PowerPC	ppc
StarCore	sc
SH	sh
Sparc	sparc
ST100	st1
TriCore	tri
V81x/V83x	810
V85x	850
x86/Pentium	386
ZSP	zsp

File Locations

MULTI searches for the register files listed in “Register Explorer Startup” on page 266, one at a time, in the following five standard directory locations unless otherwise noted. When MULTI finds one of the register files, it discontinues the search and uses the located file as the root GRD file (see “Register Explorer Startup” on page 266). MULTI searches the directories in the order listed.

1. The directory containing the program being debugged.
2. The directory containing the MULTI executable (`multi.exe`).
3. The **registers** directory located in your personal configuration directory.
4. The **registers** directory located in the site-wide configuration directory.
5. The **registers** directory located in the MULTI **defaults** directory.

For information about configuration directories, see Chapter 8, “Configuring and Customizing MULTI” in the *MULTI: Editing Files and Configuring the IDE* book.



Note MULTI does not search for *prog.grd* in the **registers** directories.



Tip If the root GRD file contains non-absolute include directives, MULTI attempts to resolve the locations of the directives by searching the directory where the root GRD file is located. If you defined the root GRD file and you want to include the MULTI installation's *cpu.grd* file in it, you can use the variable `${__MULTI_DEFAULTS_DIR__}` when specifying the path to *cpu.grd*. For example, for a PowerPC target, you could specify:

```
%include "${__MULTI_DEFAULTS_DIR__}/registers/ppc.grd"
```

Overloading Default Register Descriptions

You should usually keep your customized register definitions in the default **.rc** file for your program. If you overload one of the default files, be careful. If you do not include the appropriate default register definition file for your target, you may not be able to access any standard registers.

Setting Defaults for an Individual Program

You can add custom register definitions for an individual program by using a register definition file named after your program. Suppose you have a program *prog* that is compiled for a PowerPC target. You should add custom register definitions into a *prog.grd* register definition file in your configuration directory. This file's contents should be of the following form in GRD format.

```
# foo.grd
#
# ... description of this file ...

# Include standard PowerPC register definitions

%include "${__MULTI_DEFAULTS_DIR__}/registers/ppc.grd"

#
# add custom register descriptions here
#
```

Whenever you debug a program *prog*, MULTI will find *prog.grd* first and load it, applying the standard PowerPC register definitions from **ppc.grd** and then the custom register definitions.



Note Be sure you insert an appropriate `%include` for the target on which your process is running. Otherwise you may be unable to view the standard registers on your target.

Setting Defaults for Multiple Programs

You can provide custom register definitions that are applied whenever you debug any program compiled for a specific target. These universal modifications are useful for defining things like custom register groups that you always use. Changes that you put in one of the default register definition files for a particular target are applied when you debug any program compiled for that target.

To apply custom register definitions to all programs compiled for a specific target, follow these steps:

1. In your local configuration directory (*user_dir*\Application Data\GHS\ on Windows and *user_dir/.ghs/* on UNIX), create a directory named **registers** if it does not already exist.
2. In the **registers** directory, create a file with the name *cpu.grd*, where *cpu* is the acronym for your processor family (for example, **ppc.grd**). For a list of possible *cpu* values, see the table in “Register Explorer Startup” on page 266.
3. Use a text editor (such as the MULTI Editor) to edit the new *cpu.grd* file:
 - a. To include MULTI’s default register definition file, add:

```
%include "${__MULTI_DEFAULTS_DIR__}/registers/cpu.grd"
```

to the top of the file. *cpu* is the acronym for your processor family. If you do not insert this `%include` directive, you may be unable to view the standard registers on your target.

- b. Add your custom register definitions to the end of the file. For example:

```
# include my custom definitions
#include "my_regs.grd"
```


Using Expressions, Variables, and Procedure Calls

Chapter 12

This Chapter Contains:

- Evaluating Expressions
- Viewing Variables
- Variable Lifetimes
- Examining Data
- Wildcards
- Procedure Calls
- Macros
- MULTI Special Variables and Operators
- Syntax Checking

This chapter describes how you can type expressions into the Debugger command pane to examine and perform calculations with the values of variables in your program. It also describes how you can use MULTI variables to examine and modify the behavior of the Debugger.

Evaluating Expressions

To evaluate an expression, enter it into the Debugger's command pane. Expressions may contain:

- Symbols (for example, you can refer to variables in your program)
- Numerical and string constants
- Math and assignment (for example, you can add values together and assign values to variables in your program)
- Procedure calls
- Macros

The Debugger calculates the value of the expression and displays it. If you want to watch the value of an expression and have it reevaluated as you step through your program, you should use a Data Explorer. See “The Data Explorer Window” on page 165.

Expression Scope

Expressions are evaluated in a specific context, which affects what variables can be meaningfully used in an expression. MULTI determines the scope of an expression based on the position of the blue current line pointer (➡) and the scope rules of the language in use.

For example, suppose you are stopped inside the function `main()`:

```
int main() {  
    int foo = 3;  
    printf("Hello World! foo = %d", foo);  
    return foo;  
}
```

If the current line pointer is located on a line above the function `main()` (that is, out of its scope), and you enter the following expression in the command pane:

```
> print foo
```

MULTI prints the output:

```
Unknown name "foo" in expression.
```

However, if the current line pointer is located inside the function `main()`, the same input:

```
> print foo
```

outputs:

```
foo = 3
```

For information about variable lifetimes, see “Variable Lifetimes” on page 278.

Expressing Source Addresses

In any expression, you may want to include the address associated with a particular source location. MULTI accepts any of the following methods of expressing source addresses:

<code>#line</code>	Address of the code at line number <i>line</i> in the current file.
<code>("foo.c" # proc # line)</code>	Address of line <i>line</i> for procedure <i>proc</i> in file foo.c .
<code>proc # line</code>	Address of line <i>line</i> for procedure <i>proc</i> .
<code>("foo.c" # proc ## label)</code>	Address of label <i>label</i> for procedure <i>proc</i> in file foo.c .
<code>proc ## label</code>	Address of label <i>label</i> for procedure <i>proc</i> .

Language-Independent Expressions

The Debugger always follows the operator definitions for the current language. For example, in C, the assignment operator is one equal sign (=) and the equality comparison is two equal signs (==). On the other hand, in some languages, (Ada, for example) colon equal (:=) is the assignment operator, and one equal sign (=) is the equality comparison. This can make definition of language-independent expressions difficult. To overcome this problem, the Debugger always recognizes colon equal (:=) as the assignment operator (in addition to the correct operator in the current language) and two equal signs (==) as the comparison operator. To ensure that expressions are language-independent, these operators should be used to implement features or capabilities (such as button definitions) that may remain in operation over several source languages.

If you are debugging multiple-language applications, use syntax appropriate to the language of the source file currently displayed in the Debugger.

Language Keywords

When the Debugger evaluates expressions, it understands the following keywords for the current source language.

Language	Keywords
C	char, const, double, enum, float, int, long, long long, q15, q31, short, signed, sizeof, struct, union, unsigned, void, volatile, __accum, __fixed
C++	(All C Keywords), class, namespace, bool

Caveats for Expressions

Consider the following when constructing expressions:

- If the expression begins the same way as a Debugger command or begins with a number, put the expression in parentheses () or explicitly use the **print** or **call** command to distinguish it from the Debugger command. For example, to look at the value of the variable `c`, enter `(c)`, `print c`, or `call c`. If you just enter `c`, the Debugger will execute the `c` (continue) command. For information about the *(expr)* and **print** commands, see Chapter 9, “Display and Print Command Reference” in the *MULTI*:

Debugging Command Reference book. For information about the **call** command, see Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

- If a filename such as **class.cc** or **namespace.cc** is the same as a language keyword and is used in an expression or command, you must enclose the filename in quotation marks. For example, the **e foo.cc#2** command executes successfully but the **e class.cc#2** command does not. The second command must be **e "class.cc"#2**.
- Use **** followed by Enter to span multiple lines.
- Comments begin with **/ *** (forward slash + asterisk) and end with either a new line or *** /** (asterisk + forward slash).
- C++ style **(//)** end-of-line comments are also supported.
- If the process has not been started, the expression may only contain constants (global addresses may be considered constants, depending on your system). After the process has started, the expression may also contain in-scope variables and procedure calls.
- If the process is running, the expression may only contain constants and, if the system allows, global and static variables and their addresses.
- If the process has not been started and was linked against shared objects, expressions referring to procedures and variables located within these shared objects may not be allowed.
- There are restrictions on procedure calls and overloaded operator calls. For more information, see “Procedure Calls” on page 285.
- MULTI may not be able to parse expressions (such as casts) involving types that contain non-type template parameters.
- C++ templated functions are not supported.
- In C++, the **. *** and **-> *** (pointer to member) operators are not supported.
- In C++, casts to reference types are not supported.
- In C++, the Debugger never calls destructors while evaluating an expression.
- In C++, you must include the **enum**, **struct**, or **union** keyword when referring to an enumeration, structure, or union.
- In C++, expressions containing fully qualified function names are treated as command line procedure calls (with one exception*). To work around this behavior, leave off the types when specifying function names. If more

than one function with the given name exists—for example, `foo(int)` and `foo(char)`—a Browse window prompts you to pick one.

*Expressions containing fully qualified names are treated as expressions when used in conjunction with MULTI's **e** or **examine** command. For information about the **e** command, see Chapter 12, “Navigation Command Reference” in the *MULTI: Debugging Command Reference* book. For information about the **examine** command, see Chapter 9, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book.

- The following C/C++ macro functionality is not implemented in the expression parser:
 - Adjacent string constant concatenation
 - The stringification operator (`#`)
 - The concatenation operator (`##`)
 - Variadic macros (macros that use `...` as an argument)
- Support for AltiVec vector data types is limited:
 - You cannot perform math operations on them.
 - You cannot cast them to or from non-vector types. You can, however, convert pointers to them. For example, you can convert `void *` to `vector float *`.
 - Vector assignment is supported, but direct numerical assignment is not. For example:

```
vector unsigned int vector 1;
vector unsigned int vector 2;

vector1 = vector2; /* This will work in an expression. */
vector1 = {1,2,3,4}; /* This will not work. */
```
 - Compiler intrinsics are not supported.
- Classes defined inside functions cannot be referenced in expressions.

Viewing Variables

There are several different methods for viewing the value of a variable, including the following:

- Click the variable in the source pane to see information about the variable in the command pane. For more information, see “Viewing Information in the Command Pane” on page 150.
- Double-click the variable in the source pane.
- Use the **print** command or the **examine** command. For information about these commands, see Chapter 9, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book.
- Enter the variable name in the command pane.
- Right-click the variable and select **View Value**.

You can use any of the formats listed in the following table to unambiguously refer to a specific variable. Note that an arbitrary variable called `fly` is used in these examples. Spaces before and after the `#` symbol are optional, but the pair of straight double quotes (`" "`) around a filename is required. Local variables need to be either static or within a procedure on the stack.



Note Previous releases of MULTI did not support all of the formats listed below.

`fly`

Performs a scope search for the variable `fly`, starting at the blue arrow and proceeding outward. Local variables, local static variables, and parameters are checked first, then file static variables, global variables, and special variables.

`$fly`

Searches the list of special variables for `fly`. See “MULTI Special Variables and Operators” on page 290.

`:fly`

`::fly`

Searches for a global variable named `fly`.

```
(stack_depth # fly)
```

```
(stack_depth ## fly)
```

Uses the *stack_depth* procedure on the call stack for the scope search. This is useful if you are debugging a recursive procedure and multiple instances are on the stack. You can then pick the instance and display the value of the variable for that instance. The procedure currently containing the program counter is at stack depth 0 (zero). See the **e** command in Chapter 12, “Navigation Command Reference” in the *MULTI: Debugging Command Reference* book. Parentheses are required for these cases.

```
(stack_depth ## label # fly)
```

Local variable *fly* in the lexical block at label *label* in procedure at stack depth *stack_depth*. Parentheses are required for this case.

```
("foo.c" # proc ## label # fly)
```

Local variable *fly* in the lexical block at label *label* in procedure *proc* in file *foo.c*. Parentheses are required for this case.

```
proc ## label # fly
```

Local variable *fly* for block at label *label* in procedure *proc*.

```
("foo.c" # proc # fly)
```

Local variable *fly* for procedure *proc* in file *foo.c*. Parentheses are required for this case.

```
proc # fly
```

Local variable *fly* for procedure *proc*.

```
("foo.c" # fly)
```

Static variable *fly* in file *foo.c*. Parentheses are required for this case.

```
. (This designator is a period.)
```

Represents the result of the latest expression.

Variable Lifetimes

The Green Hills compilers augment the location description (such as register number, stack offset, memory location) for user variables with lifetime information, which indicates when the value at the given location is valid.

When you use Debugger commands (for example, **print** or **view**) or Data Explorers to evaluate expressions, the messages listed in the following table may appear next to the value of the expression.

Dead
The value displayed may be invalid because the location used to store the value of this variable may have been reused by the compiler to store the value of a temporary (or another) variable.
Not Initialized
The value displayed may represent an uninitialized value.
Optimized Away
The variable was optimized away by the compiler and does not have any storage. No value will be displayed in conjunction with this message.
Part of Expression Dead
One or more of the values used in the displayed expression may be invalid. See above for more on what this means.
Part of Expression Not Initialized
One or more of the values used in the displayed expression may be uninitialized. See above for more on what this means.

For information about expression scope, see “Expression Scope” on page 272.

Examining Data

The following sections describe commands and shortcuts for examining data. In the Debugger source pane, you can examine most items by double-clicking them. See “The Data Explorer Window” on page 165.

Variables

Variable names are represented exactly the same way they are named in the program. The case sensitivity of the current source language is used.

To display the value of a variable in the Debugger command pane, do one of the following:

- Click the variable name in the source pane.
- Enter the variable name in the Debugger command pane. If the variable has the same name as a MULTI command, preface the variable with the **print** command (for example, `print e`) or enclose the variable in a pair of parentheses, (for example, `(e)`). For information about the **print** and *(expr)*

commands, see Chapter 9, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book.

Examining Preceding Memory Locations

To view memory before the last item displayed, enter:

```
^ [count] [format]
```

where:

- *count* — Specifies the number of memory locations to print. If you do not specify a count, one memory location is printed.
- *format* — Specifies the format to be used and has the form *style[size]*, where each memory location printed has the size *size*. If a size is not specified, the size of the last item printed is used. If you do not specify a format, the format of the last item printed is used.

The *count* and *format* options should not be separated by a space.

If the last item printed was not a memory location, this command simply prints the item again with the new *count*, *style*, and *size*.

For more discussion of expression formats and their interpretations, see “Expression Formats” on page 280.



Note Examining preceding memory locations may not function as expected when displaying assembly on a target with variable length instructions.

Expression Formats

You can use an expression format to specify the output format of some commands. An example expression format follows:

```
print [/format] expr
```

An expression format *format* is of the form:

```
[count][style[size]]
```

where *count* is the number of times to apply the format style *style*, and *size* is the number of bytes to format. For example, `print /4d2 fly` prints, starting at `fly`, four 2-byte numbers in decimal. If not specified, *count* defaults to one, and *size* defaults to the size of the type printed.

In addition to a number, *size* can be specified as one of the following values:

- `b` — One byte (usually a character)
- `s` — Two bytes (usually a short)
- `l` (lowercase `L`) — Four bytes (usually an int)

These are appended to *style*. For example, `print /xb fly` prints a single character-sized hex value. Whenever a format contains a preset size (that is, character-sized in the case of `/b`, or int-sized in the case of `/O`), the size is dictated by the target (for example, a target-sized character).



Note Size specifiers override any preset size from a format. For example, `print /Ob fly` prints a byte-sized octal, not an int-sized octal.

The following table lists the options for *style*.

Value	Effect
<code>a</code>	Prints a string using <i>expr</i> as the starting address. This prints to the first null character or 128 characters, whichever happens first. The <i>size</i> value forces the printing of a given number of bytes, regardless of the occurrence of null characters.
<code>A</code>	Prints <i>expr</i> as an address, using the size of an address as its preset size.
<code>b</code>	Prints <i>expr</i> in decimal as the target's default character size.
<code>B</code>	Prints <i>expr</i> in binary.
<code>c</code> <code>C</code>	Prints <i>expr</i> as an ASCII character.
<code>d</code> <code>D</code>	Prints <i>expr</i> in decimal. If capitalized, <i>expr</i> is printed as an int-sized value.
<code>e</code> <code>E</code>	Converts <i>expr</i> to the style <code>[-] d.ddde+dd</code> , where there is one digit before the radix character and the number of digits after it is equal to the precision specification given for <i>size</i> . If <i>size</i> is not present, then the size of a double on the current target is used.

Value	Effect
<code>f</code> <code>F</code>	Converts <i>expr</i> to the decimal notation in the style <code>[-] ddd.ddd</code> , where the number of digits after the radix character is equal to the precision specification given for <i>size</i> . If <i>size</i> is not present, then the size of a double on the current target is used. If <i>size</i> is explicitly zero, then no digits or radix characters are printed.
<code>g</code> <code>G</code>	Prints <i>expr</i> in style <code>f</code> or <code>e</code> . The style chosen depends on the converted value. Style <code>e</code> is used only if the exponent resulting from the conversion is less than -4 or greater than the precision given by <i>size</i> . Trailing zeros are removed from the result. A radix character appears only if followed by a digit. This is the default for floats and doubles. If <i>size</i> is not present, then the size of a double on the current target is used.
<code>i</code>	Using the <i>expr</i> as an address, disassembles a machine instruction.
<code>I</code>	(Uppercase <code>i</code>) Using the <i>expr</i> as an address, disassembles a machine instruction. If the address maps evenly to a line number in the source, it prints the source line first. This allows you to see what the compiler generated for a line of source. Using the mixed source/assembly mode in the source pane is an easier way to view the same information. However, this format may be useful if you want to save the information to a file. For example: <pre>> tempfile print /200I myfunc > c</pre> This sequence prints the first 200 instructions of the function, <i>myfunc</i>, and saves the output to the file tempfile. See also the <code>>></code> command in “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. For other ways to save information to files, see the dumpfile and dbprint commands in Chapter 9, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
<code>n</code> <code>N</code>	Prints <i>expr</i> using the “normal” format based on its type. If no format is specified, this is the default. If capitalized, <i>expr</i> is not dereferenced; otherwise, structures and pointers are dereferenced.
<code>o</code> <code>O</code>	Prints <i>expr</i> in octal. If capitalized, <i>expr</i> is printed as an int-sized value.
<code>p</code>	(Lowercase <code>p</code>) Prints the name of the procedure containing address <i>expr</i> , along with the filename and the source line or instruction that addresses maps. If <i>size</i> is <code>1</code> or <code>b</code> , only the procedure name will be printed. If <i>size</i> is <code>2</code> or <code>s</code> , only the filename and procedure name will be printed.

Value	Effect
P	(Uppercase p) Prints the name and signature of the procedure containing address <i>expr</i> .
q	Prints the floating point number that has the same bit representation as the binary number given by <i>expr</i>. For example: <pre>> print /q 0x3ff199999999999a 1.100000</pre> If <i>expr</i> is larger than 4 bytes in size, a double is printed instead. See also <i>x</i> below.
r	Prints the bounds of a ranged type or variable of a ranged type, such as a C bitfield.
s	Prints a string using <i>expr</i> as a pointer to the beginning of the string. Same as <code>print /a *<i>expr</i></code> .
S	Prints a formatted dump of a structure. This is the default for items of type <code>struct</code> .
t	Prints the type of variable or procedure.
u U	Prints <i>expr</i> in unsigned decimal. If capitalized, <i>expr</i> is printed as an int-sized value.
v	Prints <i>expr</i> using the “normal” format based on its type. This is identical to <i>n</i> .
x X	Prints <i>expr</i> in hexadecimal. If capitalized, <i>expr</i> is printed as an int-sized value. If <i>expr</i> is a floating point number, prints the binary number that has the same bit representation as the floating point number given by <i>expr</i>. For example: <pre>> print /x 1.1 0x3ff19999_9999999a</pre> See also <i>q</i> above.
z	Prints <i>expr</i> as a set.



Note For more information about expression formats and the various commands that use them to display data or expressions, see the commands **print**, **examine**, and **eval** in Chapter 9, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book.

Language Dependencies

In C++, when the Debugger displays a class, it also displays the fields of all the parents of that class, including virtual parents, if that information is available. Static fields associated with a class are also displayed. Additionally, fields of the class or its parents that are stored by reference will be displayed as an address preceded by an at sign (@).

Wildcards

In some contexts, the Debugger supports the use of wildcards in procedure and file names (for example, with the **e** command and in the File Locator and Procedure Locator). A question mark (?) matches any single letter, while an asterisk (*) or an at sign (@) matches any number of letters. For example, the sequence ??* matches all names that are at least two characters long.

The table below lists several different formats for referring to procedures in C++. Text you substitute for the replaceable words (*italicized*) may contain wildcards.

<i>class::func</i>	Matches all members whose names match <i>func</i> of all classes whose names match <i>class</i> , regardless of their arguments.
<i>::func</i>	Matches all global or static functions whose names match <i>func</i> . Argument types may be supplied to restrict the match.
<i>func</i>	Matches all functions, whether class members or not, whose names match <i>func</i> .

Procedure Calls

From the command pane, you can call procedures in your program if the program has been linked with the library **libmulti.a**. When you set the Debugging Level to **MULTI** and then do a build, the Builder automatically links in **libmulti.a**. If you are not using the Builder, you must do one of the following to link with **libmulti.a**:

- Use the compiler command line option **-G**.
- Use the build-time command line option **-lmulti**.

For more information, see the *MULTI: Building Applications* book for your target processor family.

If you try to call a procedure and MULTI detects that **libmulti.a** was not linked in to the executable, MULTI gives an error message saying that **-lmulti** is necessary.

Expressions may contain procedure calls. For example:

```
fly = AddArgs(1, 2) * 3;
```

In C++, overloaded operators may be called provided that they are not inlined and that the left side of the expression is not contained completely within a register. Thus, the following expression:

```
complex(1,2) + complex(2,3)
```

is converted into the appropriate procedure calls. Constructors are called when appropriate, again provided that they are not inlined.

However, if `fly` were a small struct contained entirely within a register, the following expression:

```
fly += bee;
```

would fail.

You can make a command line procedure call to any non-inlined procedure in the program being debugged. For example, assume that the procedure `printf()` is referenced in the program, and thus the code for it is on the target. In this case you may enter:

```
printf("Hello, %s!\n", "world")
```



Note On many systems, due to line buffering, it is necessary to print a new line before any output appears.

If the name of the procedure you are trying to call is the same as that of a **MULTI** command, the **MULTI** command is executed instead of the procedure. To specify that **MULTI** commands are ignored when you call a procedure from the command pane, enter the **call** command before the procedure. For example, suppose you want to make a command line procedure call to a procedure named **e** that takes an integer. Enter:

```
call e(1)
```

in the command pane. If you enter:

```
e (1)
```

MULTI executes the **e** command instead. For information about the **call** command, see Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

To find out what procedures are available to be called, use one of the following commands to list the procedures in the program being debugged:

- **browse procs** (see the **browse** command in “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book)
- **lp *** (lowercase **l**, lowercase **p**, asterisk; see the **l** command in Chapter 9, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book)



Tip To gain access to library routines that are not referenced anywhere in the program code, and thus are not linked into the program image, add a dummy reference to the program and recompile.

Caveats for Command Line Procedure Calls

- Any breakpoints encountered during command line procedure calls are handled as usual.
- In some cases, interrupts may need to be disabled before command line procedure calls will work.

- If MULTI detects an executing task, it prevents you from making command line procedure calls via a freeze-mode connection to an operating system kernel. This restriction prevents MULTI from overwriting register values of non-current tasks when attempting to restore the register file after a procedure call. This could happen if a context switch occurred during the procedure call and the kernel did not completely restore the register file when switching back to the task where you initiated the call (as is common practice with floating-point registers on tasks that do not use floating point). You can bypass this restriction if you know it is safe to do so by issuing the command **configure AllowProcCallInOsaTask on**.
- If function prototype information is available, the Debugger checks the function prototype and converts each argument expression to the type of the corresponding argument. If it is not available, automatic promotion of arguments and detection of invalid arguments are not supported. In this case, you should ensure that function arguments specified are compatible with the function being called.
- When evaluating an expression, the Debugger may call any compiled function, with or without arguments, including both application and operating system functions. However, an OS function on the target system is only called if it is already linked into your program. You are responsible for linking any system calls that are called from the command line into the program.
- C++ templated functions are not supported.
- In C++ or any language with inlined procedures, a procedure that is always inlined (so there is no stand-alone version of the procedure) may not be called.
- In C++, when the expression evaluator is unable to disambiguate overloaded procedure names, a dialog box prompts you to identify what function to use.
- In C++, default arguments are not inserted.
- In C++, the class member operator `()`, the function call operator, and the `new()` and `delete()` operators are not supported.
- In C++, any procedure or method whose return type has a copy constructor cannot be called from the command line.
- In C++, any procedure or method that takes a parameter having a type with a copy constructor cannot be called from the command line.
- If you are using a run-mode debug server such as **rtserv** or **rtserv2**, you may only make command line procedure calls on halted tasks that are actually

runnable (as opposed to halted tasks that were pended before being halted). MULTI attempts to detect and prevent procedure calls to tasks that appear to have been halted while performing a system call (tasks are not runnable in this context). However, MULTI is not always able to detect situations in which it is unsafe to make command line procedure calls on a given run-mode task that is halted. If you are debugging an INTEGRITY run-mode target, see “Run-Mode Debugging” in the *INTEGRITY Development Guide* for more information.

- Command line procedure calls to functions that use the host I/O facilities of the Debugger to make blocking calls (for example, reading input from the user via the I/O pane) can cause the Debugger to appear unresponsive. In these cases, you can hit the Esc key to halt the blocked thread.

Macros

From the command pane, you can evaluate C/C++ macros defined in the program you are debugging if the program has been compiled with the Debugging Level set to **MULTI** and if the macro is used somewhere in the program.

The Debugger treats regular macros differently than macros that accept arguments. Examples of both types of macros follow.

```
#define REGULAR_MACRO 0
```

is a regular macro, whereas

```
#define FUNC_MACRO(x) (x*2)
```

is a macro with arguments.

If a regular macro and another symbol in your program share the same name, the symbol takes precedence over the macro if the symbol is local to the current function or file. If, however, a macro with arguments and another symbol in your program share the same name, the symbol always take precedence over the macro.

When you evaluate a macro, MULTI expands it regardless of your location in the program (even if you are currently viewing a file that is out of the scope of the macro definition). If multiple definitions of a macro exist across the program, the Debugger uses the definition local to the current file. If the current file

does not contain a definition, the debug information generated by the compiler designates one of the definitions in the program for consistent use by MULTI.

The following table lists the most common methods of interaction with macros.

Action	Meaning
Click	• Regular macros: displays the evaluated value of the macro, whatever that may be. In some instances, this may result in an error message similar to Unknown name "foo" in expression. This is because the Debugger is evaluating the macro in the current context, and a variable necessary to completely evaluate it is not defined within the current scope. • Macros with arguments: displays the definition of the macro.
Click and Select	• Regular macros: equivalent to a click. • Macros with arguments: displays the evaluated value of the macro, if the arguments are selected as well. If they are not, this is equivalent to a click. The caveats that apply to clicking a regular macro (see above) apply to clicking and selecting a macro and its arguments.
Right-click	Opens a shortcut menu. See "The C/C++ Macro Shortcut Menu" on page 551.
In a standard expression (that is, <i>not</i> an address expression)	Expands the macro with its evaluation in the current scope, as click above (or click and select for macros with arguments) does. Expression evaluation is aborted entirely if any of the variables used by the macro are not defined in the current scope.

MULTI Special Variables and Operators

The Debugger maintains a list of special variables and operators that are not a part of your program, but can be used in the Debugger as if they were. For example, you can use a special variable or operator in an expression that you evaluate in the Debugger.

The MULTI special variables and operators include user-defined variables (such as `$foo`), pre-defined system variables (such as `__DATA`), processor registers (such as `$r1`), and special operators (such as `$bp_addr(%bp_label)`).

When the Debugger is evaluating an expression and it finds a variable name such as `foo`, it first performs a scope search in the program to see if the variable exists. If the variable does not exist, then the list of special variables and operators is searched. Variable names beginning with a dollar sign (\$), such as `$foo`, are assumed to be special variables or operators.

User-Defined Variables

You can define a new variable by entering an appropriate statement into the Debugger command pane. The statement must adhere to the following format:

`$variable_name [=expression]`

where:

- *variable_name* should not be a reserved keyword. Reserved keywords include commands and system variables.
- The value of the optional argument *expression* initializes the variable. If you do not specify *expression*, the initial value of the variable is zero (0).

User-defined variables are of the same type as the last expression assigned. For example, entering:

- `$foo=1`

creates the special variable `$foo`, assigns it the value 1, and makes its type int.

- `$bar=3*4`

creates the special variable `$bar`, assigns it the value 12, and makes its type int.

- `$baz=myInstance.a`

creates the special variable `$baz`, assigns it the value of `myInstance.a`, and makes it the same type as `myInstance.a`.

User-defined variables are just like any other, except that it is meaningless to evaluate their addresses.

System Variables

The following table lists the currently defined system variables. Modifying the values of these variables changes the way the Debugger operates. To display the value of a system variable, enter it in the Debugger command pane with a dollar sign prepended to it (for example, `$ANSICMODE`).

ANSICMODE

When set to 0 (zero), expressions are evaluated as they are in K+R C. When set to 1, which is the default value, expressions are evaluated as in ANSI C. Generally, this affects how `unsigned shorts`, `unsigned chars`, and `unsigned bitfields` are coerced. By default in K+R, they are coerced to `unsigned int`, whereas in ANSI they are coerced to `int`. Thus `((unsigned short) 3) / -3` yields different results in ANSI and K+R. The type of `sizeof` is different, as is the interpretation of the `op=` operators in certain obscure cases.

CONTINUECOUNT

If this is 0 (zero) or 1, the Debugger will stop at the next breakpoint. If it is 2, the Debugger will stop at the second breakpoint reached by the process, and so on. Use the `c` command to set its value. For example, to set it to 3, enter: `c @3`. For information about the `c` command, see “Continue Commands” in Chapter 14, “Program Execution Command Reference” in the *MULTI: Debugging Command Reference* book.

DEBUGSHARED

Enables/disables debugging of shared objects. This system variable is only relevant with targets that support shared libraries, such as certain native UNIX platforms or advanced embedded real-time operating systems.

DEREFPOINTER

Controls whether or not pointers are automatically dereferenced when displayed by the **print**, **examine**, and **view** commands. For information about the **print** and **examine** commands, see Chapter 9, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book. For information about the **view** command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

DISNAMELEN
Controls the length of the label printed when labels appear in certain cases in disassembly. For example, the disassembly of a call or branch may include the target label; DISNAMELEN controls how many characters of that constant to print.
SERVERTIMEOUT
How long (in seconds) MULTI will wait for a debug server to respond before concluding that the server has failed. MULTI will prompt the user to close the connection or keep waiting.
TASKWIND
If this is 0 (zero), the Task Manager (for multitasking targets) will be disabled.
VERIFYHALT
Verifies halting a process before setting a breakpoint by bringing up a confirmation dialog.

System variables beginning with an underscore (`_`), such as those shown in the following table, represent the internal state of the Debugger and are not normally listed. To see them, use the command `!s _` (lowercase `L`, lowercase `s`, underscore). For information about the `!s` command, see Chapter 9, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book.

_ASMCACHE

When set to 1, which is the default value, the disassembly of program code in the Debugger window is done by reading data from the executable file, not from the program being debugged. This allows a faster disassembly display to the screen. Setting `_ASMCACHE` to 0 (zero) forces the Debugger to read the text to be disassembled from the program being debugged, instead of from the buffer or executable file. If instruction memory is modified or destroyed and `_ASMCACHE` is 1, then displays of disassembled instructions will continue to show the original, unmodified instructions in the executable file. This is confusing because the instructions actually executed are not those shown by the disassembly display.

Sometimes, when a peculiar behavior occurs on the target system, such as when the process stops on an apparently valid instruction or when it refuses to single-step or continue past a valid instruction, then the instruction memory on the target system has been corrupted. Try setting `_ASMCACHE` to 0 (zero) and redisplaying the assembly code. You may find invalid instructions at the point of failure. (You may need to turn off `_INTERLACE` (assembly) mode and examine another part of the program, turn `_INTERLACE` mode back on, and then return to the point of the entry to clear out the Debugger's internal disassembly cache.)

_AUTO_CHECK_COHERENCY

If nonzero, automatic coherency checking is enabled. If zero, automatic coherency checking is disabled. For more information, see "Detecting Coherency Errors" on page 417.

See also the `_AUTO_CHECK_NUM_ADDRS` variable below.

_AUTO_CHECK_NUM_ADDRS

Sets the number of addresses to check for coherency. Because extra memory is read on every stop, use care when setting the number of addresses to be read. Setting a large number of addresses may slow normal run control. By default, MULTI checks either 16 addresses or the number of addresses lasting the length of 4 instructions (whichever is fewer).

The value of this variable is only meaningful if automatic coherency checking is enabled. See the `_AUTO_CHECK_COHERENCY` variable above.

For more information, see "Detecting Coherency Errors" on page 417.

`_CACHE`

If non-zero, the Debugger uses a cache for reading memory from the target. The cache is invalidated every time the process state changes. This speeds up embedded debugging. Because certain devices such as hardware registers and control ports must be accessed using a specific memory access size, you must disable the `_CACHE` variable before viewing memory that corresponds to these devices.

For related commands, see the **memread** and **memwrite** commands in Chapter 11, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.

`_DATA`

Used only for position-independent data (PID) systems where the executable is linked as if it were at one address, while it runs at another. This variable is set to the offset between the location at which the data segment resides and at which it is linked. This is set on the command line with the **-data** option.

`_DISPMODE`

Determines whether assembly code is interlaced with source code in the source pane. See “The Command Pane Shortcut Menu” on page 552. This variable has no effect in **Assem** source pane display mode.

`_INIT_SP`

Tells the Debugger the value of the stack pointer at program start up, in certain target environments where this information is not available.

`_LANGUAGE`

Shows which language-specific rules for the expression evaluator are in use. 0 (zero) means C, 3 means C++, 7 means Assembly, and 31 [default] means auto-select based on the type of current file. You should not usually need to change this system variable from its default value.

`_LINES`

This controls the number of lines displayed by the **printwindow** command, and in non-GUI mode, also controls the number of lines shown between `More?` prompts. For information about the **printwindow** command, see Chapter 9, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book.

_OPCODE

If non-zero, then disassembly mode displays the hexadecimal value of the instruction.

_TEXT

Used only for position-independent code (PIC) systems where the executable is linked as if it were at one address, while it runs at another. This variable is set to the offset between the location at which the text segment resides and links. This is set on the command line with the **-text** option.

The following system variables are read-only.



Note Supported values for the system variables whose names begin with **_TARGET** are defined in the **os_constants.grd** file located at *install_dir/defaults/registers*.

_BREAK

The current breakpoint number.

_CURRENT_TASKID

The task ID of the currently executing OSA task.

_EXEC_NAME

The name and path of the program currently loaded in the Debugger.

_FILE

The name of the current file.

_INTERLACE

Indicates whether assembly code is displayed in the source window. If there is assembly code currently displayed in the source window, then this is 1; otherwise it is 0 (zero).

_LAST_COMMAND_STATUS

The status of the last MULTI command execution (0 means failure, 1 means success).

_LAST_CONNECT_CMD_LINE

The argument(s) last run with the MULTI **connect** command. For information about the **connect** command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.

_LINE

The current line number.

<code>_MULTI_DIR</code>	The name of the directory that contains the MULTI executable.
<code>_MULTI_MAJOR_VERSION</code>	MULTI's major version (for example, 5 in MULTI 5.0.6).
<code>_MULTI_MICRO_VERSION</code>	MULTI's micro version (for example, 6 in MULTI 5.0.6).
<code>_MULTI_MINOR_VERSION</code>	MULTI's minor version (for example, 0 in MULTI 5.0.6).
<code>_NONGUIMODE</code>	This indicates whether or not MULTI was started in non-GUI (-nodisplay) mode.
<code>_PID</code>	The identification number (ID) of the process, as reported by the debug server.
<code>_PROCEDURE</code>	The name of the current procedure.
<code>_PROCESS</code>	MULTI's internal slot number for the current process.
<code>_REMOTE_CONNECTED</code>	This is 1 if a remote target connection has been established, 0 (zero) otherwise.
<code>_RTSERV_VER</code>	The version number of rtserv (2 indicates rtserv2 , 1 indicates rtserv , and 0 indicates a non- rtserv connection).
<code>_SELECTION</code>	A string variable representing the current selection from the source pane.

`_STATE`

The process state, where:

- 1 = No child
- 2 = Stopped
- 3 = Running
- 4 = Dying
- 5 = Just forked
- 6 = Just exec'ed
- 7 = About to resume
- 8 = Zombied

`_TARGET`

Contains a unique identifier indicating the target processor's family and variant.

`_TARGET_COPROCESSOR`

Contains a unique identifier indicating the target coprocessor's variant.

`_TARGET_FAMILY`

The family of the target on which the program being debugged is running.

`_TARGET_IS_BIGENDIAN`

Indicates whether the target is big endian (1) or little endian (0).

`_TARGET_OS`

Displays an identifier indicating which OS is running on the target, if such information is available.

`_TARGET_MINOR_OS`

Displays an identifier indicating which minor type of OS is running on the target, if such information is available.

`_TARGET_SERIES`

This may contain information about whether the target processor is a member of a certain series of processors (for example, the PowerPC 400-series). This does not work for all processor families.

`_TASK_EXIT_CODE`

The exit code of the current task.

Processor-Specific Variables

Processor registers are included as predefined variables. To find out what register names are available on your system, use the command **l r** (lowercase L, lowercase R) to list the registers. For information about the **l** command, see Chapter 9, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book.

All registers can be treated as integers of the correct size for the register. Be careful when you modify the contents of registers when you are debugging high-level code; the results of these modifications can often produce unpredictable effects.

The following predefined Debugger internal variable is also included.

`$result`

Holds the return value of the most recently completed procedure. This variable is an alias for the register on the processor architecture used for returning integers. On most systems, this is also the register to return pointers. It may be written to as well as read from.

This value is not guaranteed to be correct; it depends on the return type of the function and the processor architecture. The actual return variable may be located elsewhere. As with any register, you must be careful when you change its value.

Special Operators

The Debugger maintains a list of special operators that are not a part of your program, but can be used in the Debugger as if they were. For example, you can use special operators in expressions that you evaluate in the Debugger.

`$bp_adr(%bp_label)`

Returns the address of the breakpoint with label *bp_label*.

`$ent(procedure)`

Deprecated. Use `$entadr(procedure)`.

`$entadr(procedure)`

Returns the address of the first instruction after the given procedure *procedure*'s prologue code. (The procedure prologue code is also known as the stack frame setup code or the entry code.)

If *procedure* is not given, the current procedure is used.

`$exists(symbol)`

`$M_sym_exists(symbol)`

Returns a Boolean indicating whether the symbol *symbol* exists within the program currently being debugged.

If you specify a keyword incorrectly in *symbol*, this operator may not return `true` or `false`. If you enclose *symbol* in quotation marks, it always returns `true` or `false`.

`$in(procedure)`

Returns `true` if the current process is stopped and the current program counter (PC) is in the given procedure *procedure*.

`$M_called_from(level, "string")`

Returns a Boolean indicating `true` if the *level* is 0, and "*string*" is the name of a function on the current call stack. Otherwise, it returns `true` if "*string*" is the name of the function at the call stack depth indicated by *level*.

`$M_can_read_address(num)`

Returns `true` if the address indicated by *num* is readable; returns `false` otherwise.

`$M_file_exists("file")`

Returns a Boolean indicating whether the file *file* exists within the program currently being debugged.

`$M_get_temp_memory(size)`

Returns an address that points at a chunk of memory of *size* address units. If *size* is too large, an error message is printed and `-1` is returned as the address. This chunk of memory is guaranteed to be valid only until MULTI needs more temporary memory on the target. In practice, it should remain valid until the next time the user stores a string or other data structure to the target (by calling a constructor at the command pane, for example).

Using this special operator requires that your program be linked with the library **libmulti.a**. For more information, see "Procedure Calls" on page 285.

`$M_offsetof(type, field)`

`offsetof(type, field)`

Returns the offset in bytes of the member field *field* within the aggregate type *type*. The first parameter must be either an aggregate (struct, union, or class) type name or an instance of an aggregate type. The second parameter must be the name of a field within that type.

Note that entering `offsetof(type, field)` in the command pane may execute a macro or procedure call if the current program has a macro or procedure call with the name `offsetof`. To guarantee the use of the Debugger's built-in operator, use `$M_offsetof()`.

<code>\$M_sec_begin("section")</code>
Returns the address of the beginning of the section <i>section</i> . It returns -1 if the section does not exist within the program currently being debugged.
<code>\$M_sec_end("section")</code>
Returns the address of the end of the section <i>section</i> . It returns -1 if the section does not exist within the program currently being debugged.
<code>\$M_sec_exists("section")</code>
Returns a Boolean indicating if the section <i>section</i> exists within the program currently being debugged.
<code>\$M_sec_size("section")</code>
Returns the size of the section <i>section</i> . It returns -1 if the section does not exist within the program currently being debugged.
<code>\$M_strcmp(expr, "string")</code>
Returns an integer greater than, equal to, or less than zero if the string pointed to by the first parameter is greater than, equal to, or less than the string pointed to by the second parameter. The first parameter must be an expression which evaluates to a string and the second parameter must be a string constant.
<code>\$M_strcpy(expr, "string")</code>
Returns the number of bytes written. The first parameter must be an expression which evaluates to a string and the second parameter must be a string constant.
<code>\$M_strprefix(expr, "string")</code> <code>\$M_strcaseprefix(expr, "string")</code>
Returns a Boolean indicating whether the first parameter is prefixed by the second (<code>\$M_strcaseprefix</code> is case-insensitive). The first parameter must be an expression that evaluates to a string and the second parameter must be a string constant.
<code>\$M_strstr(expr, "string")</code>
Returns a pointer to the first occurrence of the second parameter string within the first parameter string, or a null pointer if it is not found. The first parameter must be an expression which evaluates to a string and the second parameter must be a string constant.
<code>\$M_typeof(var)</code>
Returns the type of <i>var</i> , suitable for use in casting. For example: <code>\$foo = ((\$M_typeof(my_var)) 0x10000)</code> would give the value of 0x10000 to the MULTI local variable <code>\$foo</code> and give it the same type as <code>my_var</code> .

```
$M_var_is_valid(var)
```

Returns true if and only if the indicated variable exists in the current scope, is initialized, and is not yet out of scope (that is, dead). Returns false otherwise.

```
$ret (procedure)
```

Deprecated. Use `$retadr (procedure)`.

```
$retadr (procedure)
```

Returns the address of the first instruction of the given procedure *procedure*'s epilogue code. (The procedure epilogue code is also known as the stack frame release code or the return code.)

If *procedure* is not given, the current procedure is used.

```
sizeof (variable)
```

```
sizeof (type)
```

```
sizeof ("string")
```

```
sizeof ((expression))
```

Behaves similarly to the C `sizeof` operator and returns the size in bytes of the type of *variable*, of *type*, or of the type resulting from *expression*. If *"string"* is specified, this operator returns `strlen("string") + 1`. Unlike the C `sizeof` operator, this operator actually evaluates any argument provided, such that if an expression with side effects is specified, the side effects occur.

Syntax Checking

The syntax checking mechanism checks the validity of a command without actually executing it, and thus without requiring target interactions and without changing the system settings.

The Debugger command **sc** performs syntax checking. It can be used in two different ways.

To check the syntax of a single command, enter:

```
sc "command"
```

To check the syntax of an entire script file and all nested files, enter:

```
sc < script_file_name
```

For more information, see the **sc** command in “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

Syntax checking is also automatically invoked whenever a breakpoint with an associated command or condition is created. The validity of the commands associated with the breakpoint is checked in the context that would exist if the breakpoint were hit. If a syntax error is found in the breakpoint command, an error message is issued and the breakpoint is not created.



Note You can use the **bpSyntaxChecking** configuration option to disable this automatic syntax checking. For more information, see the **bpSyntaxChecking** option in “The More Debugger Options Dialog” in Chapter 9, “Configuration Options” in the *MULTI: Editing Files and Configuring the IDE* book.

For example, entering the command:

```
sc "print abcdef"
```

will echo the error message:

```
Syntax Checking:  Unknown name "abcdef" in  
expression.
```

and entering the command:

```
b main { print abcdef; }
```

will echo the error messages:

```
Syntax Checking:  Unknown name "abcdef" in  
expression.
```

```
Failed to set software breakpoint owing to syntax  
error.
```


Using the Memory View Window

Chapter 13

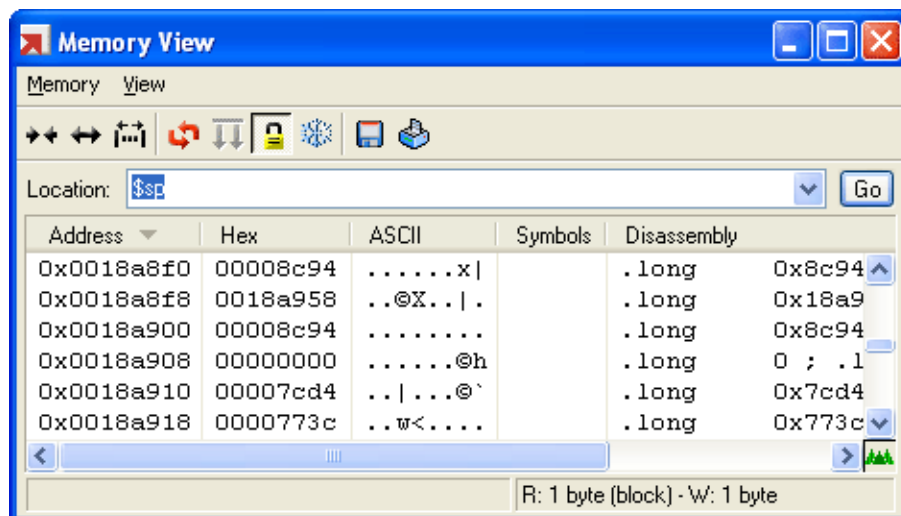
This Chapter Contains:

- Setting the Active Location
- The Memory Pane
- Controlling Memory Access
- Editing Memory
- The Memory View Toolbar
- Memory View Menus

The **Memory View** window is useful for examining large buffers, strings, and other contiguous blocks of memory. The window can be configured to display memory in a variety of formats. Memory may also be modified from this window.

To open the **Memory View** window, perform one of the following actions:

- Click the **Memory** button (M) located on the toolbar.
- Select **View** → **Memory**.
- In a Data Explorer, select **Tools** → **Memory View on “variable”** to open a **Memory View** window examining the same memory location as the Data Explorer.
- In the command pane, enter the **memview** command. For information about this command, see Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.



Setting the Active Location

The active location for the **Memory View** window is displayed in the **Location** bar at the top of the window, and the row containing this location is highlighted in the memory pane.

When you specify a new active location, target memory is read and the displayed memory begins with the address of the active location. To change the active location, do one of the following:

- Enter an address expression, a section name, or a section name and offset (`.text+0x400`, for example) in the location bar, and press Enter.
- Select a previously entered location from the location bar's drop-down list.



Note Scrolling the window does not change the highlighted location or the contents of the **Location** bar. To return to the active location, press Enter.

Locking the Current Symbol's Location

By default, if you enter a symbol as the active location, MULTI sets the memory pane location to the address for the symbol at the time you entered it. If the address for the symbol changes, the text in the location bar turns red, but the address highlighted in the memory pane does not change. To display and change the active location to the new address, press Enter.

If you would like the **Memory View** window to track the location of the symbol each time it changes, click the **Lock Address** button () so that it no longer appears pushed down. In this mode, the active location is updated every time the address for the symbol changes.

The Memory Pane

The memory pane is composed of rows of memory displayed in an address column and multiple view columns. To change the number of bytes displayed in each row, use the following buttons:

- **Contract** () and **Expand** () — Change the byte-width of the rows by a factor of two each time they are pressed.
- **Set Row Width** () — Opens a window in which you can specify the width of the rows.

The **Address** column displays the location of the first byte of each row and can never be removed from the pane. You can view the rows of memory in ascending or descending order. To change the order, click the header of the **Address** column. Each view column displays the formatted contents of memory at that location.

Formatting View Columns

You can independently format each view column in the memory pane. To format a view column, do one of the following:

- Right-click the header of a column and select a formatting option from the shortcut menu.
- Select **View** → **Column** *number name* and select a formatting option from the submenu.

The primary formatting options are **Hex**, **Decimal**, **Binary**, **Floating Point**, **Fixed Point**, **Ascii**, **Symbols**, **Disassembly**, and **Offset**. You may see additional view column formats if your CPU supports them.

In addition to primary formatting options, secondary formatting options may appear if the selected primary format supports them. For the comprehensive list of options, see the “The Column Submenu” on page 314.

The following information is specific to select formats.

- **Numerical formats** — Any memory that cannot be read is displayed as a series of dashes.
- **Ascii** format — Any memory that cannot be represented by an ASCII character is displayed as a dot (.).
- **Symbols** format — Symbols that continue from previous addresses are represented with the string “ . . . ”.
- **Disassembly** format — In regions of memory without symbol information, inaccurate instructions may be displayed if the beginning of the window is not aligned with the start of an instruction.
- **Big Endian** and **Little Endian** format — If the selected byte order is not the default order for your target, it is displayed in the column header.

Managing View Columns

The following table summarizes the actions you can perform on view columns:

Action	Procedure
Add Column	One of: • Select View → Add New Column → <i>format</i>. • Right-click the header beyond the right edge of the last column and select a format from the shortcut menu. • Right-click the header of a column and select Add New Column from the shortcut menu.
Remove Column	One of: • Right-click the header of a column and select Remove from the shortcut menu. • Select View → Column number name → Remove.
Hide Column	One of: • Right-click the header of a column and select Hide from the shortcut menu. • Select View → Column number name → Hide.
Show Column	Select View → Column number name → Show .

Action	Procedure
Resize Column Automatically	One of: • Right-click the header of a column and enable Size to Fit from the shortcut menu. • Enable View → Column <i>number name</i> → Size to Fit.
Reorder Columns	Drag the header of a column to its new location.

Controlling Memory Access

You can control how the **Memory View** window accesses memory by freezing the window, setting access sizes, or disabling block reads. The following sections describe each of these actions.

Freezing the Memory View Window

The contents of the memory pane are automatically updated each time the target halts or steps. To prevent this update, click the **Freeze** button (). Location changes and automatic updates are disabled until the window is unfrozen. To unfreeze the window, click the **Freeze** button again, and the contents of the window are updated. To force the window to update its contents, even if frozen, by immediately re-reading target memory, do one of the following:

- Click the **Reload** button (.
- Right-click the memory pane and select **Reload from Target**.
- Select **Memory** → **Reload from Target**.

Setting Access Sizes

The preferred read and write access sizes are shown in the *access size display area*, located in the right side of the status bar. The **Memory View** window defaults to these sizes when accessing target memory. To change the access sizes, do one of the following:

- Double-click the access size display area. In the **Access Sizes** dialog box that appears, select the new access sizes from the drop-down lists.
- Right-click the access size display area, select **Read Access Size** or **Write Access Size**, and select the new access size from the submenu.

- Select **Memory** → **Set Access Sizes**. In the **Access Sizes** dialog box that appears, select the new access sizes from the drop-down lists.

The contents of the memory pane are automatically refreshed to reflect your changes.



Note Access sizes do not affect how memory is displayed in the view columns.

Disabling Block Reads

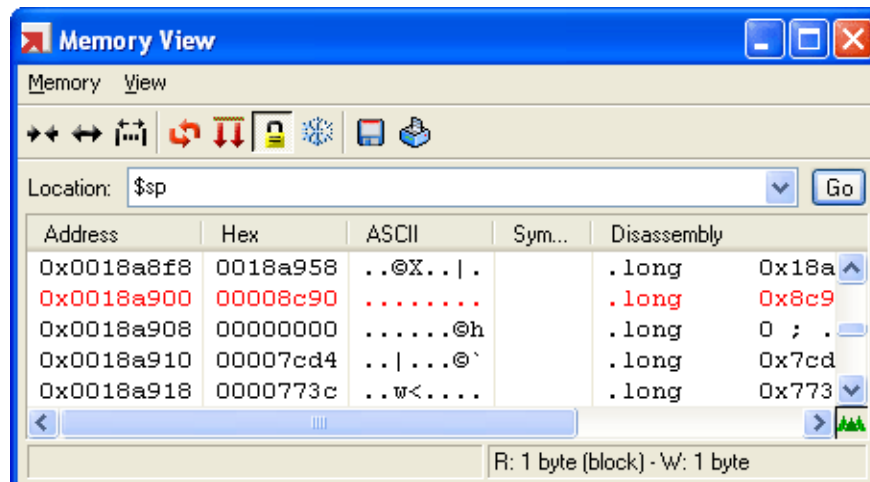
To increase performance, the **Memory View** window performs all target memory reads as block memory accesses. If you would like to set an access size because of target or memory requirements, but your debug server prevents you from doing so, disable the **Use Block Reads** option. The **Memory View** window then performs all reads as individual memory accesses of the preferred read size. To disable the **Use Block Reads** option, do one of the following:

- Clear **Memory** → **Use Block Reads**.
- Right-click the access size display area, and clear **Use Block Reads**.

Editing Memory

You can edit the contents of memory in the **Hex**, **Decimal**, **Binary**, and **ASCII** columns. To edit the contents of memory, perform the following steps:

1. Click the character you want to edit.
2. Type a new value. All columns update to reflect your change, which is displayed in red lettering.



3. Repeat steps 1 and 2 for all of the characters you would like to modify.
4. Do one of the following:
 - Click .
 - Right-click the memory pane and select **Write Changes to Target**.
 - Select **Memory** → **Write Changes to Target**.

Until you perform this step, your changes will not be written to target memory.

The Memory View Toolbar

The **Memory View** toolbar contains buttons that allow you to control the **Memory View** window and perform memory-specific operations. The following table describes the buttons in the **Memory View** window.

Button	Effect
 Contract	Contracts the byte-width of the rows by a factor of two each time it is pressed.
 Expand	Expands the byte-width of the rows by a factor of two each time it is pressed.
 Set Row Width	Opens a window in which you can specify the width of the rows.

Button	Effect
 Reload	Forces the window to update its contents, even if frozen, by immediately re-reading target memory. Clicking this button is equivalent to selecting Reload from Target from the Memory menu or the shortcut menu.
 Write Changes to Target	Writes memory modifications made in the Memory View window to the target. Clicking this button is equivalent to selecting Write Changes to Target from the Memory menu or the shortcut menu.
 Lock Address	Locks the address of the symbol in the location bar. This is the default behavior. To unlock the address and track the location of the symbol each time it changes, click this button so that it no longer appears pushed down. For more information, see “Locking the Current Symbol’s Location” on page 305.
 Freeze	Prevents location changes and automatic updates. To re-enable these window updates, click the Freeze button again.
 Save Current View to File	Saves the contents of the Memory View window to a text file. The information contained in the text file is formatted exactly as it appears in the columns of the Memory View window.
 Print	Prints the memory pane contents. All view column data is printed for each address currently in view.
 Close	Closes the Memory View window. Clicking this button is equivalent to selecting Memory → Close .

Memory View Menus

The following sections describe the menu items available from the **Memory View** window.

The Memory Menu

The following table lists the options available in the **Memory** menu.

Item	Meaning
Copy	Opens a window that allows you to copy one region of memory to another. This is equivalent to the copy -gui command.
Fill	Opens a window that allows you to fill a specified region of memory with the given value. This is equivalent to the fill -gui command.
Find	Opens a window that allows you to search memory for a specified value. This is equivalent to the find -gui command.
Compare	Opens a window that allows you to compare two regions of memory. This is equivalent to the compare -gui command.
Load	Opens a window that allows you to load the contents of a file on the host machine into a portion of memory on the target. This is equivalent to the memload -gui command.
Dump	Opens a window that allows you to save a region of memory on the target to a file on the host. This is equivalent to the memdump -gui command.
Write Changes to Target	Writes memory modifications made in the Memory View window to the target. Selecting this menu item is equivalent to clicking the Write Changes to Target button () or selecting Write Changes to Target from the shortcut menu.

Item	Meaning
Reload from Target	Forces the window to update its contents, even if frozen, by immediately re-reading target memory. Selecting this menu item is equivalent to clicking the Reload button () or selecting Reload from Target from the shortcut menu.
Set Access Sizes	Opens a window that allows you to specify preferred read and write access sizes. Selecting this menu item is equivalent to double-clicking the access size display area.
Use Block Reads	Toggles whether or not the Memory View window performs all target memory reads as block memory accesses. By default, this option is enabled. For more information, see “Disabling Block Reads” on page 309. Selecting this menu item is equivalent to right-clicking the access size display area and selecting Use Block Reads .
Save View to File	Saves the contents of the Memory View window to a text file. The information contained in the text file is formatted exactly as it appears in the columns of the Memory View window. Selecting this menu item is equivalent to clicking the Save Current View to File button ()
Print	Prints the memory pane contents. All view column data is printed for each address currently in view. Selecting this menu item is equivalent to clicking the Print button ()
Save Settings as Default	Saves the memory pane settings as the default Memory View window configuration.
Close	Closes the Memory View window. Selecting this menu item is equivalent to clicking the Close button ()

For information about the commands referenced in the preceding table, see Chapter 11, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.

The View Menu

The following table lists the options available in the **View** menu.

Item	Meaning
Add New Column	Opens a submenu providing a number of view column formats. Selecting a format adds a view column of the specified type to the memory pane. New columns are added to the right of the last column in the window. For more information about view formats, see the “The Column Submenu” on page 314.
Column <i>number name</i>	Opens a submenu with a number of options affecting the specified column. See the “The Column Submenu” on page 314. This menu item is repeated for each column. Selecting this menu item is equivalent to right-clicking a column’s header.

The Column Submenu

The following table lists the options available from the **View → Column *number name*** submenu. You can also access these same menu items by right-clicking a column’s header. You may see additional view column formats if your CPU supports them.

Unless otherwise stated, all descriptions of toggle options explain the behavior of the option when enabled.

Item	Meaning
Hide/Show	Hides or shows the column. This option toggles between Hide and Show .
Remove	Deletes the column.
Size to Fit	Sets the column to automatically adjust its width to contain displayed data.
Hex	Displays the column’s memory contents as hexadecimal numbers.*
Decimal	Displays the column’s memory contents as decimal numbers.*
Binary	Displays the column’s memory contents as binary numbers.*

Item	Meaning
Floating Point	Displays the column's memory contents as floating-point numbers.*
Fixed Point	Displays the column's memory contents as fixed point numbers.*
Ascii	Displays the column's memory contents as ASCII characters. Any memory that cannot be represented by an ASCII character is displayed as a dot (.).
Symbols	Displays the column's memory contents as a comma-separated list of global symbols. Symbols that continue from previous addresses are represented with the string " . . . ".
Disassembly	Displays the column's memory contents as a semicolon-separated list of assembly instructions. In regions of memory without symbol information, inaccurate instructions may be displayed if the beginning of the window is not aligned with the start of an instruction.
Offset	Displays the offset between the active location and the start of the row.
number byte(s)	Sets the column's unit size. Available <i>numbers</i> are 1, 2, 4, and 8. These menu items are only available for the Hex , Decimal , and Binary columns.
Signed	Interprets memory data as signed decimal integers. This menu item is only available for the Decimal column.
Unsigned	Interprets memory data as unsigned decimal integers. This menu item is only available for the Decimal column.
Single Precision	Interprets memory data as single precision floating point numbers. This menu item is only available for the Floating Point column.
Double Precision	Interprets memory data as double precision floating point numbers. This menu item is only available for the Floating Point column.

Item	Meaning
q15	Interprets memory data as q15 fixed point numbers. This menu item is only available for the Fixed Point column.
q31	Interprets memory data as q31 fixed point numbers. This menu item is only available for the Fixed Point column.
Big Endian	Sets the column byte order to big endian. If this is not the default byte order for your target, the setting is displayed in the column header. This menu item is only available for the Hex, Decimal, Binary, Floating Point, and Fixed Point columns.
Little Endian	Sets the column byte order to little endian. If this is not the default byte order for your target, the setting is displayed in the column header. This menu item is only available for the Hex, Decimal, Binary, Floating Point, and Fixed Point columns.

*Numerical formats display unreadable memory as a series of dashes.

The Shortcut Menu

The following table lists the options available in the shortcut menu that appears when you right-click the memory pane.

Item	Meaning
Set Hardware Breakpoint	Opens a submenu that allows you to set a hardware breakpoint on the memory address you right-clicked. For a description of the items available in this submenu, see “The Set Hardware Breakpoint Submenu” on page 317. This menu item is only available for the Address column.

Item	Meaning
Write Changes to Target	Writes memory modifications made in the Memory View window to the target. Selecting this menu item is equivalent to clicking the Write Changes to Target button () or selecting Memory → Write Changes to Target .
Reload from Target	Forces the window to update its contents, even if frozen, by immediately re-reading target memory. Selecting this menu item is equivalent to clicking the Reload button () or selecting Memory → Reload from Target .

The Set Hardware Breakpoint Submenu

The following table lists the options available from the shortcut menu's **Set Hardware Breakpoint** submenu. This submenu is only available for the **Address** column.

Item	Meaning
Set Write Hardware Breakpoint	Sets a new hardware breakpoint that is hit when your program writes to the memory address you right-clicked.
Set and Edit	Opens the Hardware Breakpoint Editor with the memory address that you right-clicked already filled in. This allows you to configure the properties of the hardware breakpoint before it is set.

Viewing Memory Allocation Information

Chapter 14

This Chapter Contains:

- Debugging Memory Allocation Errors
- Using the Memory Allocations Window

This chapter explains how you can use the MULTI IDE to collect memory allocations information and debug memory allocation errors.

Debugging Memory Allocation Errors

It can be very difficult to trace bugs involving the misuse of the standard library functions `malloc()` and `free()`, because the effects of these bugs may not be noticed immediately.

For example, if you mistakenly free memory that was not allocated with `malloc()`, or try to use memory that has already been freed, your process may behave in unpredictable ways.

MULTI supports two types of run-time error checking to catch these kinds of bugs: *general memory allocation checking* and *intensive memory allocation checking*.



Note For embedded targets, run-time error checking must be enabled by setting Builder or driver options before you compile your program, as described in “General Memory Allocation Checking” on page 320 and “Intensive Memory Allocation Checking” on page 321. For some native targets (Solaris and x86-based Linux), you may be able to access run-time error checking information simply by opening the **Memory Allocations** window before starting execution of your program (see page 325).

General Memory Allocation Checking

This form of run-time error checking will increase your heap size considerably and reduce your process speed slightly. It requires less overhead than intensive memory allocation checking (see “Intensive Memory Allocation Checking” on page 321). It generates run-time errors when:

- Memory that has not been allocated with `malloc()` is subsequently freed
- The same area of memory is freed twice
- Recently freed memory is written to

In addition, general memory allocation checking allows you to trace *memory leaks*, which can occur when memory that is allocated with `malloc()` is not subsequently freed. In this situation, your process may unexpectedly run out of

memory at some indeterminate future time. For more information about tracing memory leaks, see “Using the Memory Allocations Window” on page 325.

To enable general memory allocation checking, set the **Debugging→Run-Time Memory Checks** option to **General (Allocations Only)** (`-check=alloc`) before compiling your program.

Intensive Memory Allocation Checking

This form of run-time error checking provides the most comprehensive `malloc()` checking. However, it will appreciably increase the size of your program code, and slow your execution speed more than general memory allocation checking (see “General Memory Allocation Checking” on page 320). It generates run-time errors when:

- Memory that has not been allocated with `malloc()` is subsequently freed
- The same area of memory is freed twice
- Recently freed memory is written to
- Attempts are made to dereference `NULL`
- Accesses are attempted past the end of a heap-allocated data structure, causing array overflow errors.

To enable intensive memory allocation checking, set the **Debugging→Run-Time Memory Checks** option to **Intensive** (`-check=memory`) before compiling your program.



Note Intensive memory allocation checking may be impractical for users with memory constraints. You may want to enable general memory allocation checking for your entire project, and use intensive checking only for particular modules where `malloc()`-related errors are most likely to occur. Alternatively, you can manually change the intensity level of your memory checking with library functions, as described in the next section, or by using the **Checking** menu of the **Memory Allocations** window (see “Using the Memory Allocations Window” on page 325).

Using Memory Allocation Library Functions

MULTI provides library functions that can control the amount and frequency of consistency checking performed by the `malloc()` library.

These functions are only available if you build your code with general memory allocation checking or intensive memory allocation checking enabled (see “General Memory Allocation Checking” on page 320 or “Intensive Memory Allocation Checking” on page 321). To call them, you must also include the **ghs_malloc.h** header file, located in *install_dir/ansi*, which declares the following two functions:

- `int malloc_set_debug_options(int newOptions)` — Sets the current intensity level and returns the old intensity level.
- `void malloc_check_lib(int useTheseOptions)` — Performs an immediate check of the `malloc()` library using the indicated intensity level.



Note If you call `void malloc_check_lib()` with 0 or `M_CHECK_LEVEL_NONE`, the function uses the intensity level set when you last called `malloc_set_debug_options()`.

Both of the preceding functions take a constant that corresponds to an intensity level. The constants are defined in **ghs_malloc.h**. For a list of the available intensity levels, see “Performing Selective Run-Time Error Checking” on page 323.

For a demonstration of the use of these error checking functions, see Example 1 on page 324.



Tip You can achieve the same functionality as calling `malloc_set_debug_options()` by selecting an intensity level in the **Checking** menu of the **Memory Allocations** window. For more information, see “Performing Selective Run-Time Error Checking” on page 323.

Performing Selective Run-Time Error Checking

As an alternative to enabling constant memory allocation checking, which may be impractical for users with memory constraints, you can enable certain memory checks in one of the following ways:

- By using the library functions described in “Using Memory Allocation Library Functions” on page 322.
- By selecting an intensity level from the **Checking** menu in the **Memory Allocations** window.



Note Performing either of the preceding actions requires that you build your code with general memory allocation checking or intensive memory allocation checking enabled. See “Using Memory Allocation Library Functions” on page 322.

The library functions take in an intensity level (corresponding to a menu item in the **Checking** menu), which controls the amount of self-checking, and the frequency of that checking, that the memory library performs on calls to `malloc()`, `calloc()`, `realloc()`, and `free()`. The following list describes the intensity levels (and corresponding menu items):

- `M_CHECK_LEVEL_NONE` (**Disable Checking**) — Does not perform additional checking.
- `M_CHECK_LEVEL_MINIMAL` (**Minimal Checking**) — Performs minimal checking on each API call.
- `M_CHECK_LEVEL_OCCASIONALLY_MAXIMAL` (**Occasionally Maximal Checking**) — Performs minimal checking on each API call, and occasionally performs full checking. This is the default setting.
- `M_CHECK_LEVEL_FREQUENTLY_MAXIMAL` (**Frequently Maximal Checking**) — Performs minimal checking on each API call, and frequently performs full checking. Checks more frequently (and with more associated overhead) than `M_CHECK_LEVEL_OCCASIONALLY_MAXIMAL`.
- `M_CHECK_LEVEL_MAXIMAL` (**Maximal Checking**) — Performs full checking on each API call. This may severely degrade the performance of your application.

Increasing the intensity level increases the amount and frequency of checking while also increasing the associated overhead and decreasing the speed of the application.

The default intensity level (`M_CHECK_LEVEL_OCCASIONALLY_MAXIMAL`) should be suitable for most applications. However, after you have narrowed down the area in which a memory error (corruption, for example) is occurring, it may be useful to increase the setting to `M_CHECK_LEVEL_MAXIMAL` shortly before the error is expected to occur.

Example 1. Enabling and Disabling All Checks

This example requires that you set the **Debugging→Run-Time Memory Checks** option to **General (Allocations Only)** (`-check=alloc`) before compiling your program.

Note that the following code is only meant as an example. Because of the errors included for demonstration purposes, it may not function exactly as documented if you copy it into your executable. Line numbers are added for clarity.

```
#include <ghs_malloc.h>

int main()
{
5   int *arr, *arr2;

    malloc_set_debug_options(M_CHECK_LEVEL_MAXIMAL); /* turn on all checks */
    arr = malloc(100*sizeof(int));
    free(arr);
10   arr[5] = 0; /* error - writing to a freed memory location. */

    /* it will check and find the last error now */
    arr = malloc(100*sizeof(int));
    free(arr);
15   malloc_set_debug_options(M_CHECK_LEVEL_NONE); /* turn off all checks */

    arr = malloc(100*sizeof(int));
20   free(arr);
    arr[5] = 0; /* error - writing to a freed memory location. */

    arr2 = malloc(100*sizeof(int));
    malloc_check_lib(M_CHECK_LEVEL_MAXIMAL); /* check malloc data structure now;
25                                     detect error from line 21 */
    free(arr2);
    return 0;
}
```

This example does the following:

- `malloc_set_debug_options(M_CHECK_LEVEL_MAXIMAL)` on line 7 enables all the `malloc()` checks, so that the error on line 10 is detected when `malloc()` is called on line 13.
- `malloc_set_debug_options(M_CHECK_LEVEL_NONE)` on line 16 disables all the `malloc()` checks, so that the error on line 21 is not detected when `malloc()` is next called at line 23.
- `malloc_check_lib(M_CHECK_LEVEL_MAXIMAL)` on line 24 performs an immediate check on the `malloc()` data structure, and detects the error on line 21.

Using the Memory Allocations Window

If you built or linked your program using the MULTI Builder, you can use the **Memory Allocations** window to examine your program's heap allocations. The **Memory Allocations** window is most useful for tracking down memory leaks, displaying overall heap usage, visualizing heap fragmentation, and detecting run-time memory errors.

To get the most out of the **Memory Allocations** window when you are developing for an embedded target, set the **Debugging→Run-Time Memory Checks** build option to **General (Allocations Only) (-check=alloc)** or to **Intensive (-check=memory)** before compiling your program. This causes your program to use debug versions of `malloc()` and `free()` that record a small call stack for each allocation, helping you to track down memory leaks. It also enables run-time memory checks to detect common memory errors like double freeing and writing to unallocated memory. For more information, see “General Memory Allocation Checking” on page 320 and “Intensive Memory Allocation Checking” on page 321. For Solaris and x86-based Linux targets, you may be able to access run-time error checking information without setting build options. Unless you have enabled call count profiling, simply open the **Memory Allocations** window before starting program execution. If you have enabled call count profiling by setting the **Debugging→Profiling - Call Count** option to **On (-p)** or **On with Call Graph (-pg)**, you must explicitly turn on memory allocation checking in the manner described above.



Note If you are using run-time error checking with an INTEGRITY project, shared libraries must be disabled. In the top-level project file, set the **Target→Operating System→Shared Libraries** build option to **No Shared Libraries (-non_shared)**. If you have disabled this option as described and still receive compile-time errors when building with run-time memory checking enabled, the option may be enabled in another project file. Make sure that no project file sets the **Target→Operating System→Shared Libraries** build option to **Link With Shared Libraries (-call_shared)**.

You can open the **Memory Allocations** window at any time by doing one of the following:

- Select **View → Memory Allocations**.
- Enter the **heapview** command in the command pane. For more information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.



Note The **Memory Allocations** window is designed for use during run-mode debugging. However, the window may also display helpful information if launched on the master process during a freeze-mode debugging session (that is, if the master process is selected in the target list when you open the window). The **Memory Allocations** window is not accessible if you attempt to open it when an OSA task is selected in the target list. For information about freeze-mode debugging, the master process, and OSA tasks, see Chapter 19, “Freeze-Mode Debugging and OS-Awareness”.

Before you can perform any of the operations available through the **Memory Allocations** window, your process must be halted, and, if you are debugging in run mode, you must be able to run the program; it cannot be blocked or pending. A convenient way to halt the process is to set a breakpoint on the last line of your program’s source code and then open the window when the process hits the breakpoint. For information about run-mode debugging, see Chapter 18, “Run-Mode Debugging”.



Note When you use the **Memory Allocations** window to view allocations, leaks, and run-time errors, code that may allow interrupts to be serviced is executed on the target. If any interrupt servicing results in calls to memory allocation routines, the memory checking code may crash or give inaccurate results. You may need to disable interrupts before using these memory analysis

operations. Use of the visualization should be safe, however, as it only reads the target memory and does not execute any code on the target.

The **Memory Allocations** window contains the following three tabs:

- **Visualization** — Displays an interactive visualization of the program’s heap and information about the application’s overall memory usage. For more information, see “Viewing the Memory Allocation Visualization” on page 328.
- **Leaks** — Displays unreachable blocks of memory. For more information, see “Viewing Memory Leaks and Allocation Information” on page 332.
- **Allocations** — Displays all allocated blocks of memory, including the unreachable blocks displayed on the **Leaks** tab. For more information, see “Viewing Memory Leaks and Allocation Information” on page 332.

By default, the **Visualization** tab is displayed when the **Memory Allocations** window is first opened (unless you have used the **showleaks**, **showallocations**, or **showerrors** argument with the **heapview** command). For information about the **heapview** command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.



Note The deprecated **findleaks** command provided in previous versions of MULTI is equivalent to **heapview showleaks**.

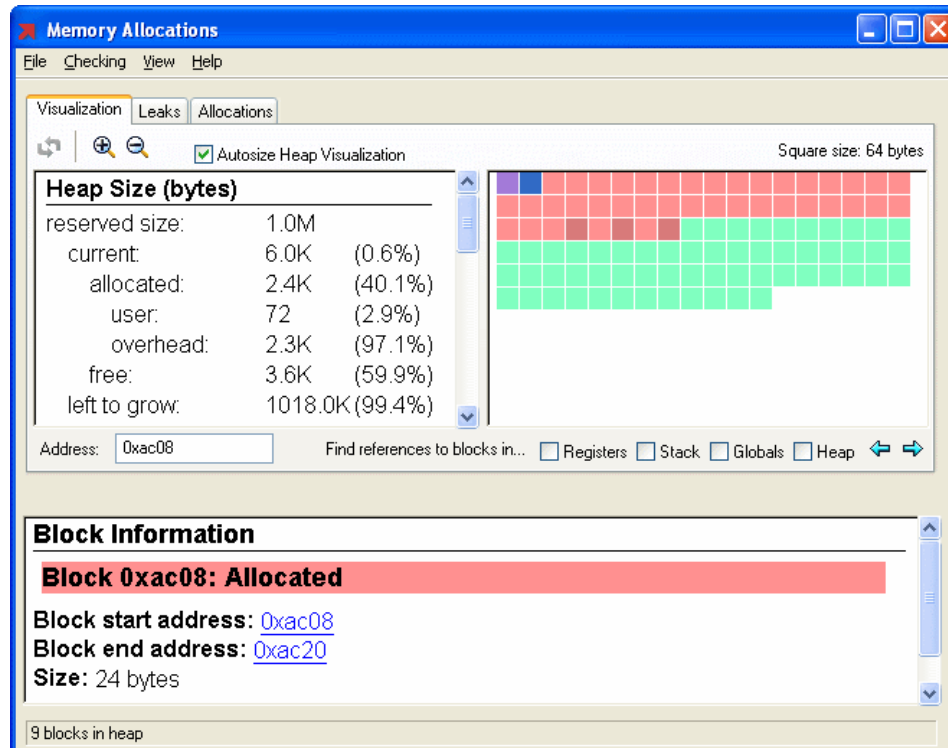
The **Memory Allocations** window contains four menus: **File**, **Checking**, **View**, and **Help**. The options on these menus are described in the table below. Unless otherwise stated, all descriptions of toggle options explain the behavior of the option when enabled.

File → Save <i>information</i>	Opens a Save Report dialog that allows you to specify a file to save the contents of the active tab (<i>information</i> will appear on the menu as View , Leaks , or Allocations , depending on which tab is active).
File → Print <i>information</i>	Prints the contents of the active tab (<i>information</i> will appear on the menu as Statistics , Leaks , or Allocations , depending on which tab is active).
File → Close	Closes the Memory Allocations window.

Checking → Disable Checking Checking → Minimal Checking Checking → Occasionally Maximal Checking Checking → Frequently Maximal Checking Checking → Maximal Checking	Specifies the amount and frequency of run-time error checking. For more information, see “Performing Selective Run-Time Error Checking” on page 323.
View → Human Readable Numbers	Displays rounded numbers in the Memory Allocations window.
View → Show Hexadecimal Values	Displays the hexadecimal equivalent for memory sizes in the Memory Allocations window.
View → Display Heap Discontinuities as Blocks	Displays unused sections of the heap as memory blocks. Otherwise, discontinuities are collapsed into a single bar.
View → Clear Runtime Errors	Clears the run-time errors displayed in the lower pane of the Memory Allocations window.
Help → Memory Allocations Help	Opens the MULTI Help Viewer on documentation for the Memory Allocations window.

Viewing the Memory Allocation Visualization

To view a visualization of an application’s memory usage, click the **Visualization** tab of the **Memory Allocations** window (if it is not already the active tab) and click the **Refresh Visualization** button (). An example **Visualization** tab view is shown next.



The **Visualization** tab of the **Memory Allocations** window is divided into several panes.

The upper-left pane displays general statistics about the memory usage of your application. The information that is available varies depending upon whether your application was built with run-time memory checking enabled. The following list describes the information that may be available in this pane.

- **Heap Size (bytes):**

- **reserved size** — Amount of memory initially reserved for the heap in bytes.
- **current** — Current size of the heap given to the application by the operating system.



Note The current heap size may exceed the reserved size if the `malloc()` library extends the heap beyond the region initially reserved for it. If this occurs in an INTEGRITY project, the additional heap space is taken from an unused block of virtual addresses in the `VirtualMemoryRegion` pool.

- **allocated** — Aggregate amount of heap memory currently used by your application in allocated blocks. This is further broken down into:
 - user** — Amount of heap memory allocated by the user. This information is only present when run-time memory checking is enabled.
 - overhead** — Amount of memory used internally by the `malloc()` library for memory error checking. This information is only present when run-time memory checking is enabled.
- **free** — Aggregate amount of heap memory available to your application in free heap blocks.
- **left to grow** — Amount of reserved memory that is left for the heap to grow. This information is not present if the heap size exceeds the reserved size and extends into unreserved memory.
- **Blocks:**
 - **total** — Total number of blocks in the heap visualization.
 - **allocated** — Number of blocks in the heap allocated by the application.
 - **free** — Number of free blocks in the heap.
 - **discontinuities** — Number of discontinuities in the heap section. Discontinuities can appear when the heap extends into unreserved memory.
- **Calls** — Number of calls to the memory management functions **sbrk()**, **malloc()**, **calloc()**, **realloc()**, and **free()**. The **sbrk()** function is typically called internally by the **malloc()** library to request memory from the operating system. These functions are only present when run-time memory checking is enabled.
- **Other Statistics:**
 - **block alignment** — All heap blocks are a multiple of the alignment for the architecture currently being debugged.
 - **cumulative allocated** — Total number of bytes the application has allocated up to the current point in its execution. Because this statistic counts only allocations, it never decreases during the course of execution. This information is only present when run-time memory checking is enabled.

- **peak user allocated** — Maximum aggregate amount of memory the application has allocated at any one time. This information is only present when run-time memory checking is enabled.
- **largest allocated block size** — Size of the largest block of memory currently allocated. This information is only present when run-time memory checking is enabled.
- **largest free block size** — Size of the current largest free block of memory. This information is only present when run-time memory checking is enabled.

The upper-right pane of the **Visualization** tab displays a visualization of the heap. In this visualization, sequences of red squares indicate allocated blocks of memory, while sequences of green squares indicate free blocks of memory. Adjacent allocated or free blocks alternate shades for clarity. Blocks that are recognized as `malloc()` overhead are marked in purple. Discontinuities in the heap are shown as long gray bars unless you have enabled **View → Display Heap Discontinuities as Blocks**, in which case, discontinuities are displayed as gray blocks. By default, the visualization is sized to fit best in the window, but you can use the zoom buttons ( and ) to look at specific sections in more or in less detail.

Click any of the squares in the heap visualization to get more details about a specific heap block. These details appear in the bottom pane, described next. You can navigate among heap blocks by clicking them in the visualization or by using the arrows located in the middle-right side of window.

The bottom pane of the **Visualization** tab displays specific details about the heap block you have chosen in the heap visualization pane. In addition, the check boxes located above the bottom pane allow you to search several locations in memory for references to a particular heap block. The following list describes the memory locations you can search.

- **Registers** — Search the general purpose registers for references to this heap block.
- **Stack** — Search the program stack for references to this heap block.
- **Globals** — Search the program globals (such as those defined in `.data` or `.bss` sections) for references to this heap block.
- **Heap** — Search the other allocated blocks of the heap for references to this heap block.



Tip Searching for references is primarily useful when you do not want to deal with the overhead of building your program with the run-time memory checking library. All the functionality of the heap visualization can be experienced without run-time memory checking enabled.

If run-time errors are detected while the **Memory Allocations** window is open, they are displayed below the block references. To clear run-time errors, select **View → Clear Runtime Errors**.

Updating the Visualization

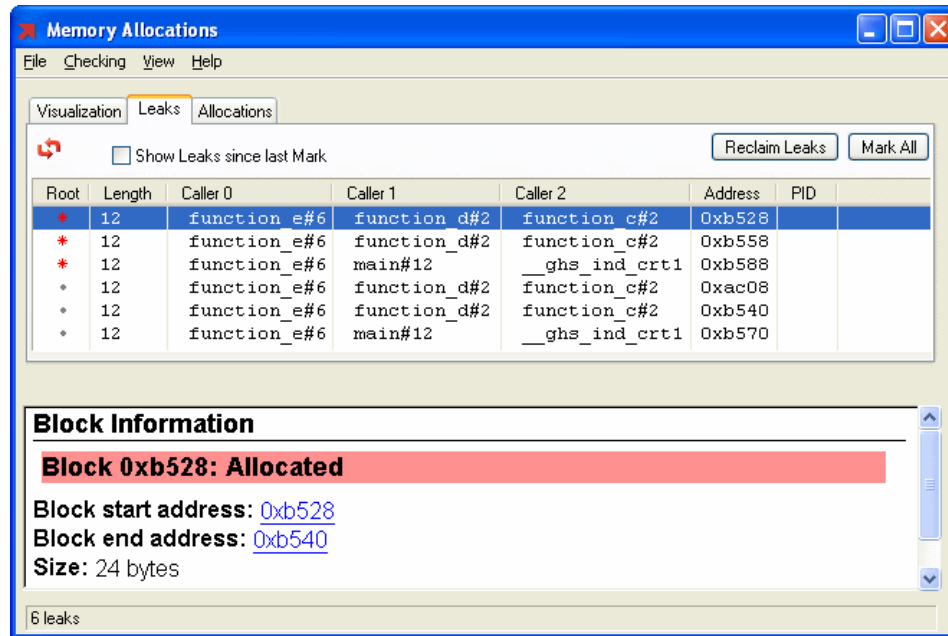
Because gathering heap visualization information can take a long time, the **Visualization** tab is not automatically refreshed when your program's state changes. A warning message appears, however, and you can click the **Refresh Visualization** button (🔄) to refresh the visualization and the statistics. Switching among the tabs does not refresh the visualization.



Note Updating the **Visualization** tab while the program is halted inside `malloc()` library code may cause incorrect heap data to be displayed.

Viewing Memory Leaks and Allocation Information

The **Leaks** tab of the **Memory Allocations** window displays unreachable blocks of memory, while the **Allocations** tab displays all allocated blocks of memory, including unreachable blocks. An example **Leaks** tab view is shown next.



The columns of the **Leaks** and **Allocations** tabs display the following information for each block of memory:

- **Root** — Any memory block that is referenced by another block in the heap is marked with a dot in this column. If a memory block is not referenced by anything else on the heap, it is considered a root and is marked with a red asterisk. In cases where a memory block contains pointers to memory that become unreachable when the block itself becomes unreachable, this marking makes it easier to find the origin of memory leaks. For example, suppose a linked list is leaked. The memory block that contains the head of the linked list is marked with a red asterisk, and everything following the head of the list is marked with a gray dot.

This information is only present when run-time memory checking is enabled.

- **Length** — The size of the allocated block in bytes.
- **Caller 0, Caller 1, etc.** — The procedure and line number (or address) of the first five levels of the call stack of the allocation. (On some native targets (Solaris and x86-based Linux), there may be more than five levels.)
- **Address** — The address of the block of memory.
- **PID** — The process ID of the process that called `malloc()`, if applicable. This information is only present when run-time memory checking is enabled.

If the **Leaks** or **Allocations** tab is up to date with respect to the **Visualization** tab, block information for the selected leak or allocation (including the references selected in the **Visualization** tab) is shown in the bottom pane.

Manipulating Procedures and Memory Chunks

You can perform the following actions on the information that is displayed on the **Leaks** and **Allocations** tabs of the **Memory Allocations** window:

To	Do this
View a memory chunk in a Memory View window	Double-click the address of the memory chunk.
Display a procedure in the Debugger's source pane	Click the procedure.
Edit a procedure in the Editor	Double-click the procedure.
Reclaim some leaks by calling <code>free()</code> on them	Select the desired leaks and click the Reclaim Leaks button on the Leaks tab. Note: This causes a procedure call to be executed on the target, so be sure the process is halted in a safe location. For more information, see "Before Performing Operations that Execute Procedure Calls" on page 334.

Before Performing Operations that Execute Procedure Calls

Some of the operations available from the **Leaks** and **Allocations** tabs of the **Memory Allocations** window cause a procedure call to be executed on the target. As a result, you should be sure that the process is halted in a safe location before performing one of these operations, which are identified in the following list:

- Refreshing leaks or allocations (see "Refreshing Leaks and Allocations Information" on page 335)
- Reclaiming leaks (see "Manipulating Procedures and Memory Chunks" on page 334)
- Tracking leaks or allocations (see "Tracking Leaks and Allocations" on page 336)

A convenient way to halt your process in a safe location is to set a breakpoint in the program and wait for the breakpoint to be hit.

The following information, which is specific to particular debugging environments, describes locations where it is unsafe to be halted when performing one of the preceding operations.

- On INTEGRITY, the program should not be halted inside a Task, such as the Idle Task, that is *not* C library enabled. Tasks that are not C library enabled are created via the `CreateTask()` kernel call. In most systems, each AddressSpace's Initial Task and all user-defined Tasks *are* C library enabled and are therefore safe to perform operations from.
- On INTEGRITY, the program should not be halted inside a Task that holds the lock from `__ghsLock()`. Additionally, no Task in the AddressSpace should be halted in a location where it holds the lock from `__ghslock()`. The preceding operations, which rely on being able to obtain this lock, become pended if you try to complete them while the lock from `__ghsLock()` is held.
- When run-time memory checking is enabled, the program should not be halted inside `malloc()` library code. The preceding operations cause `malloc()` library code to be run on the target, but the library is not re-entrant. Additionally, if you refresh leaks or allocations, a small amount of dynamic memory is required. To guarantee heap integrity, ensure that the program is halted outside of `malloc()` library code and that the target will not exhaust the heap.

Refreshing Leaks and Allocations Information

Because the operation of finding leaks and allocations can be slow and invasive, the **Leaks** and **Allocations** tabs do not automatically refresh every time the process is stopped. To refresh the leaks and allocations information, click the **Refresh Leaks** or **Refresh Allocations** button (🔄). Note that refreshing leaks or allocations causes a procedure call to be executed on the target, so be sure that the process is halted in a safe location. For more information, see “Before Performing Operations that Execute Procedure Calls” on page 334.



Note Because tasks running in the same INTEGRITY AddressSpace share memory, refreshing leaks or allocations information for one task while other tasks in the same AddressSpace are still running can lead to inconsistent results. If MULTI detects that other tasks in the AddressSpace are still running, a dialog

box appears to inform you of this and to give you the opportunity to cancel the refresh operation. In this situation, we recommend that you do the following:

1. Click **No** to cancel the refresh operation.
2. Set breakpoints in your code to stop each of the other tasks. Note that it is important to stop each of the tasks in your code, and not in kernel code or in other shared library code. Failing to stop the tasks in your code can cause the refresh operation to become pended, waiting for system resources to be released.
3. Wait until all your tasks have halted at the breakpoints you set.
4. Refresh leaks or allocations information for the original task.

Tracking Leaks and Allocations

You can track memory leaks and allocations while your process executes by using the **Mark All** button, which marks all blocks of memory currently allocated in the heap, and the **Show Leaks since last Mark** or **Show Allocations since last Mark** check box, which toggles whether only unmarked (or new) leaks or allocations are shown. Marked leaks are shaded gray in the **Leaks** and **Allocations** tabs. To monitor memory leaks and allocations:

1. Halt the application at a safe and useful checkpoint. For information about safe locations, see “Before Performing Operations that Execute Procedure Calls” on page 334.
2. Click the **Refresh Leaks** or **Refresh Allocations** button () to update the **Leaks** or **Allocations** information.
3. While still halted in a safe location, click the **Mark All** button on the **Leaks** or **Allocations** tab of the **Memory Allocations** window. This marks the current set of allocations.
4. Resume execution of the application.
5. Halt the application in a safe location again.
6. Click the **Refresh Leaks** or **Refresh Allocations** button () to update the **Leaks** or **Allocations** information.
7. Select the **Show Leaks since last Mark** or **Show Allocations since last Mark** check box so that the tabs display only those leaks and allocations that occurred since the last time you marked the allocations.
8. Repeat this process as necessary as you run through the entire application.



Note The **Mark All** button and the **Show Leaks since last Mark** and **Show Allocations since last Mark** check boxes are only available when run-time memory checking is enabled.



Tip You can achieve the same results by using Debugger commands in the command pane. The command **heapview setmark** marks the current set of allocations, and the command **heapview showleaks -new** or **heapview showallocations -new** displays the new leaks or allocations. For more information about the **heapview** command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

Caveats for Leak Information

- The **Leaks** tab of the **Memory Allocations** window may not display all of the leaks in the program being debugged at any given moment. If dead variables are still on the stack or in registers, MULTI assumes they are alive and does not report anything they reference as a leak. Once the process has run further (for example, returned out of the current routine, which should clear out the stack), some or all of these false negatives vanish and additional leaks may show up.
- In some target implementations, a register may be a base register, a general purpose register, or both. It is possible that the value of the base register may point into the heap (for example, on some targets, when the register is the SDA base pointer, and there is no data). This can prevent a memory leak from being reported for a dead object at the same location as the base register.
- Blocks referenced only via pointer arithmetic may be falsely marked as leaks. Additionally, false positives may occur in INTEGRITY applications where thread local storage is allocated from the MemoryPool, such as for tasks declared in the **.int** file or created with `CreateProtectedTask()` or `CommonCreateTask()`. Multithreaded INTEGRITY applications may also generate false positives if the thread stack comes from the MemoryPool.

Collecting and Viewing Profiling Data

Chapter 15

This Chapter Contains:

- Types of Profiling Data
- Overview of Profiling Methods
- Collecting Profiling Data
- Caveats for Profiling
- Disabling the Collection of Profiling Data
- Viewing Profiling Information
- The Profile Window
- Viewing Profiling Information in the Debugger
- Performing Range Analyses
- Managing Profiling Data
- Using the protrans Utility

MULTI's powerful profiling capabilities allow you to gather and view information about the performance and coverage of your programs. You can use this information to analyze and optimize the efficiency of your code.

Types of Profiling Data

MULTI supports three basic types of profiling data: *program counter (PC) samples*, *call count data* (with and without call graph support), and *coverage analysis data*. Each are described below:

- PC samples — Contain data about where the process spends its time. You can view how much time is spent in each function, each basic block, each source line, each assembly instruction, and the entire program. You may be able to make execution faster by using this information to improve code that accounts for an unnecessarily large percentage of the total execution time.
- Call count data
 - With call graph support — Contains a record of how many times each function is called, as well as which child functions are called by each parent function and how many times each child is called.
 - Without call graph support — Contains a record of how many times each function is called.
- Coverage analysis data — Contains a profile of basic block executions. If a given basic block is never executed, the group of instructions making up the block is considered dead code for the given sample input. Since the application never reaches the dead code, you can either remove the code from the application or try to discover if any functionality is missing from the application as a result of it.

For a general description of the methods used to collect profiling data, see the next section.

Overview of Profiling Methods

Both the method that MULTI uses to collect profiling data and the types of data that MULTI collects is determined by what you choose to profile. You can profile all tasks in your system (if you are using INTEGRITY version 10 or later), a trace-enabled target, a run-mode task or AddressSpace on an INTEGRITY target, or a stand-alone program. The following list states what types of data MULTI outputs for each (see “Types of Profiling Data” on page 340).

- If you profile all tasks in your system (requires INTEGRITY version 10 or later) — INTEGRITY periodically samples the PC of running tasks, and the run-mode debug server (**rtserv2**) delivers the following profiling data to MULTI:

- PC samples

MULTI uses the samples to display information for all tasks in your system. For more information, see “Profiling All Tasks in Your System” on page 447.

- If you profile a trace-enabled target — MULTI converts trace data into profiling data that is much more complete, in terms of which instructions are represented, than profiling data generated by sampling methods. (Sampling methods are usually timer-driven and can miss important hot spots in a process.)

MULTI is able to convert collected trace data into all the following types of profiling data:

- Program counter (PC) samples
 - Call count with call graph data
 - Coverage analysis data
- If you profile a run-mode task or AddressSpace on an INTEGRITY target, or if you profile a stand-alone program — MULTI collects one or more of the following types of data:
 - PC samples
 - Call count data with or without call graph support
 - Coverage analysis data



Note Only PC samples provide task-specific profiling data. You cannot collect call count or coverage analysis profiling data for a specific task—only for the encompassing AddressSpace.

Generating Profiling Data for a Task, AddressSpace, or Stand-Alone Program

Green Hills simulators, run-mode debug servers, and some native debug servers can sample the PC without instrumenting your code. If you are using a debug server that does not support PC sampling, or if you want to be able to analyze additional types of profiling data, you can compile your program with one or more of the following Builder or driver options. For information about setting Builder and driver options, see Chapter 2, “The MULTI Project Manager” in the *MULTI: Building Applications* book and Chapter 4, “The Compiler Driver” in the *MULTI: Building Applications* book, respectively.

Each of the following options instruments your code to generate profiling data for a run-mode task or AddressSpace on an INTEGRITY target, or for a stand-alone program. The type of instrumentation performed, which differs for each option, determines what type of profiling data is generated. For information about the instrumentation performed for each of the following options, see Chapter 5 in the *MULTI: Building Applications* book. For information about profiling data, see “Types of Profiling Data” on page 340.

- To obtain PC samples — Compile with the **Debugging→Profiling - Target-Based Timing** option set to **On** (or with the **-timer_profile** compiler driver option).

It may be necessary to link additional custom support code with your embedded application to obtain PC samples. If your target supports target-based timing profiling, you can find more information in “Target-Based Timing Profiling” in Chapter 5 in the *MULTI: Building Applications* book for your target processor family.

Note that the **-timer_profile** option is only applicable if you are profiling a non-INTEGRITY application over a freeze-mode connection to an embedded target. This option does not apply if you are profiling a simulated, native, run-mode, or INTEGRITY target.



Note To obtain PC samples for a Linux native application, you must link with the **libmulti.a** library.

- To obtain call count data with call graph support enabled — Compile with the **Debugging→Profiling - Call Count** option set to **On with Call Graph** (or with the **-pg** compiler driver option).

OR

To obtain call count data without call graph support enabled — Compile with the **Debugging→Profiling - Call Count** option set to **On** (or with the **-p** compiler driver option).



Note Do not attempt to pass both the **-p** and **-pg** compiler driver options.

- To obtain coverage analysis data — Compile with the **Debugging→Profiling - Coverage** option set to **On** (or with the **-a** compiler driver option).

Support for profiling, the options and steps necessary to collect profiling data, and the exact methods used to collect profiling data vary depending on your target environment. For a full description of the profiling options available for your target and for any additional target-specific information about profiling, see Chapter 5 in the *MULTI: Building Applications* book and Chapter 6, “Builder and Driver Options” in the *MULTI: Building Applications* book.

Collecting Profiling Data

This section contains detailed instructions for collecting profiling data for all tasks in your system, for a trace-enabled target, a single run-mode task or AddressSpace on an INTEGRITY target, or a stand-alone program. For a general overview of the methods used to collect profiling data, see “Overview of Profiling Methods” on page 341.

The scope of profiling is limited to what you select in the target list *prior* to enabling the collection of profiling data. If you select a run-mode connection and then enable profiling, profiling data is collected for all tasks in your system. If you select a trace-enabled target, profiling data is collected for everything on the target. If you select a run-mode task or AddressSpace on an INTEGRITY target, or if you select a stand-alone program, profiling data is collected for the task (PC samples only), AddressSpace, or stand-alone program. If you collect profiling data for one of these items and then want to profile another, you must

disable profiling collection and clear existing profiling data before selecting and collecting data for the second item.



Note If you enable profiling from the Trace List or PathAnalyzer, or with the **tracepro** command or the **TimeMachine** → **Profile** menu selection, profiling data is generated for all traced tasks or AddressSpaces, regardless of the selection in the target list. (For information about the **tracepro** command, see Chapter 20, “Trace Command Reference” in the *MULTI: Debugging Command Reference* book.)

Collecting Profiling Data for All Tasks in Your System

To collect profiling data for all tasks in your system (requires INTEGRITY version 10 or later), perform the following steps:

1. In the Debugger’s target list, select the run-mode connection.
2. Enable the collection of profiling data and open the **Profile** window by doing one of the following:
 - Select **View** → **Profile**.
 - In the Debugger command pane, enter the **profile** command. For information about the **profile** command, see Chapter 13, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.

By default, profiling is enabled when you open the **Profile** window. Do not close the **Profile** window until you are finished viewing profiling information. (Closing this window halts the collection of profiling data and clears existing profiling data.) For information about the **Profile** window, see “The Profile Window” on page 348.

3. In the target list, select a task to view profiling data for that task in the **Profile** window.

For information about the type of data that appears in the **Profile** window, see “Overview of Profiling Methods” on page 341.

In addition to the profiling information that appears in the **Profile** window, the Debugger’s target list displays processor usage for all tasks in your system. For more information, see “Profiling All Tasks in Your System” on page 447. For information about other ways to view collected profiling data, see “Viewing Profiling Information” on page 347.

Collecting Profiling Data for a Task, AddressSpace, or Stand-Alone Program

To collect profiling data for a run-mode task or AddressSpace on an INTEGRITY target, or to collect profiling data for a stand-alone program, perform the following steps:

1. Before compiling your program, determine whether it is necessary to set Builder options (or corresponding compiler driver options) to enable profiling. For information, see “Generating Profiling Data for a Task, AddressSpace, or Stand-Alone Program” on page 342.
2. If necessary, compile your program with the appropriate profiling options set.
3. Connect to your target.
4. In the Debugger’s target list, select the run-mode task, run-mode AddressSpace, or stand-alone program that you want to profile.
5. Enable the collection of profiling data and open the **Profile** window by doing one of the following:
 - Select **View → Profile**.
 - In the Debugger command pane, enter the **profile** command. For information about the **profile** command, see Chapter 13, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.

By default, profiling is enabled when you open the **Profile** window. Do not close the **Profile** window until you are finished viewing profiling information. (Closing this window halts the collection of profiling data and clears existing profiling data.) For information about the **Profile** window, see “The Profile Window” on page 348.

6. Step or run the program.
7. Usually, you should let your program run at least once before expecting to see any information in the open **Profile** window. (By default, MULTI processes collected profiling data automatically each time your program exits. After the data has been processed, profiling information appears in the **Profile** window.) In some situations, however, you may prefer to halt the process early and manually dump and process collected profiling data. For more information, see “Manually Dumping Profiling Data” on page 367 and “Manually Processing Profiling Data” on page 368.

For information about the types of data that appear in the **Profile** window, see “Overview of Profiling Methods” on page 341.

For information about the different ways you can view collected profiling data, see “Viewing Profiling Information” on page 347.

Caveats for Profiling

- If you are profiling a run-mode task on an INTEGRITY target, only PC samples provide task-specific profiling data. You cannot collect call count or coverage analysis profiling data for a specific task—only for the encompassing AddressSpace.
- If you profile a stand-alone program, you can only collect PC samples if you open the **Profile** window prior to downloading, stepping, or running the program.

Disabling the Collection of Profiling Data



Note This information is only relevant if you are profiling all tasks in your system, a run-mode task or AddressSpace on an INTEGRITY target, or a stand-alone program.

To disable the collection of profiling data:

- If you are profiling all tasks in your system — Halt the target. Alternatively, click the **Close** button (✗) to close the **Profile** window, disable the collection of profiling data, and clear existing profiling data. (Whether this button appears on your toolbar depends on your MULTI configuration.)
- If you are profiling a run-mode task or AddressSpace, or if you are profiling a stand-alone program — In the **Profile** window, click the **Stop collecting profile information** button (🛑). Alternatively, click the **Close** button (✗) to close the **Profile** window, disable the collection of profiling data, and clear existing profiling data. (Whether this last button appears on your toolbar depends on your MULTI configuration.)

Viewing Profiling Information

After you have collected profiling data (see “Collecting Profiling Data” on page 343), you can view the information in a variety of ways:

- *In profiling reports* — The tabs in the **Profile** window display profiling information in different report formats. For more information, see “Profiling Reports” on page 350.
- *In the Debugger window* — The Debugger can indicate the percentage of time spent in each source line (or each instruction, if in assembly or interlaced assembly display mode), which lines of code were never executed, the number of times each source line was executed, or the number of times each function was called. For more information, see “Viewing Profiling Information in the Debugger” on page 363.
- *In a Range Analysis window* — This window allows you to specify a range of hexadecimal addresses to examine. For more information, see “Performing Range Analyses” on page 364.

- *In a dynamic call graph Browse window* — This window displays a call graph generated at run time. For more information, see “Browsing Dynamic Calls by Function” on page 226.

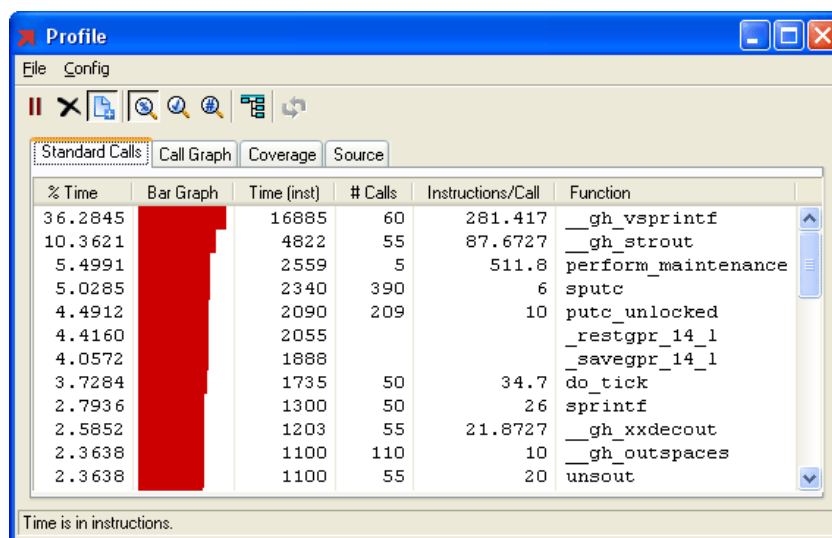


Note Support for these display capabilities depends on what you profile and/or on the type of profiling data available. For more information, see the referenced sections.

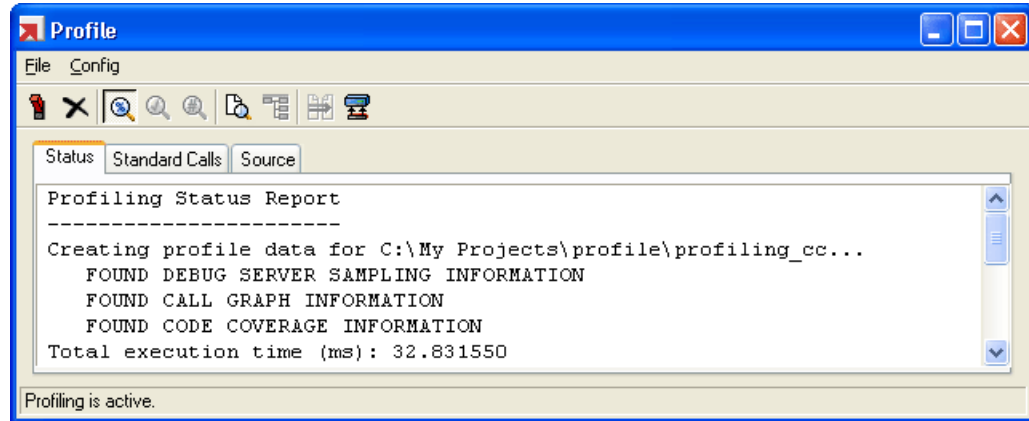
The Profile Window

The **Profile** window allows you to view collected profiling data in various report formats. For information about how to enable profiling and open the **Profile** window, see “Collecting Profiling Data for All Tasks in Your System” on page 344 or “Collecting Profiling Data for a Task, AddressSpace, or Stand-Alone Program” on page 345.

If you profile all tasks in your system or if you profile a trace-enabled target, the **Profile** window looks similar to the following graphic.



If you profile a run-mode task or AddressSpace on an INTEGRITY target, or if you profile a stand-alone program, the buttons on the toolbar and the availability of reports and of menu items differ. In this situation, the **Profile** window looks similar to the following graphic.



Note The information available in the **Profile** window may vary slightly according to your target type. For example, profiling with a cycle accurate simulator may yield additional columns in the reports containing cycle information. For additional notes about profiling for your specific target, see the *MULTI: Configuring Connections* and the *MULTI: Building Applications* books for your target type.

The following sections explain the various features of the **Profile** window. For a detailed description of all the menu items and buttons, see “The Profile Window Reference” on page 357.

Continual Updates in the Profile Window



Note This information is only relevant if you profile all tasks in your system, or if you profile a trace-enabled target.

When the following conditions are met, the **Profile** window continually updates select information:

- If you are profiling all tasks in your system — The window continually updates processor usage per function (that is, the standard calls report).

To pause the continual updating of information, click the **Pause updating of profile display** button () in the **Profile** window.

Profiling Reports

Each of the tabs in the **Profile** window represents a profiling report. Depending on what profiling data is available, there can be up to six profiling report tabs in the **Profile** window: **Status**, **Standard Calls**, **Call Graph**, **Coverage**, **Block Detailed**, and **Source**. To view any one of these reports in the **Profile** window, simply click the corresponding tab. (If the data required for a particular report was not collected, no tab appears for that report.) You can also view each profiling report by entering the following command in the Debugger command pane:

```
profilereport report
```

where *report* specifies one of the report types and can be:

- `status` — See “Status Report” on page 351.
- `calls` — See “Standard Calls Report” on page 352.
- `graph` — See “Call Graph Report” on page 353.
- `coveragesummary` — See “Coverage Report” on page 354.
- `coveragedetailed` — See “Block Detailed Report” on page 355.
- `sourcelines` — See “Source Report” on page 356.

Each profiling report generally consists of several columns of information and a status bar at the bottom of the report. You can sort most of the columns by clicking the column header. To sort in the opposite direction, click the same header again. To resize a column, drag the separator to the left or right.

By default, reports omit C++ qualifiers when displaying function names. You can view fully qualified function names by hovering your mouse pointer over a function, or you can display fully qualified function names in the reports by selecting **Function Names** → **Long** from the **Config** menu. To change back to the default behavior, select **Function Names** → **Short**.

Within each of the reports, single-clicking a function or other component causes the Debugger to jump to that function or component in the source pane. Double-clicking opens a MULTI Editor window on the function or component (if appropriate). Right-clicking opens a shortcut menu. For a description of the shortcut menu items, see “The Procedure Shortcut Menu” on page 548.

You can save, append, or print any profiling report by selecting a menu item or by using the **profilereport** Debugger command:

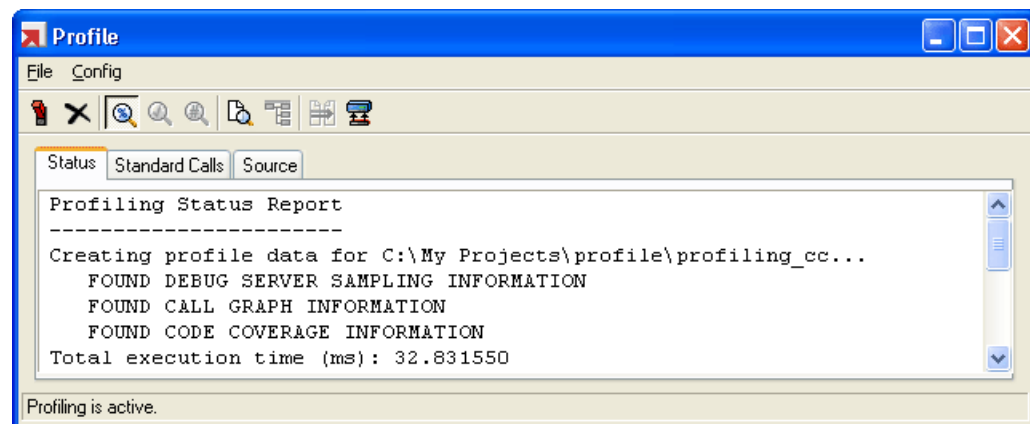
- To save the text of the report that is currently displayed in the **Profile** window, select **File** → **Save Report** or use the **profilereport save** command. The default file extension given to the saved text file is **.rep**.
- To append the text of the currently displayed report to a previously created file, select **File** → **Append Report** or use the **profilereport append** command.
- To print the text of the report that is currently displayed, select **File** → **Print Report** or use the **profilereport print** command.

For more information about the **profilereport** command, see Chapter 13, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.

Each of the profiling reports is described in greater detail in the following sections.

Status Report

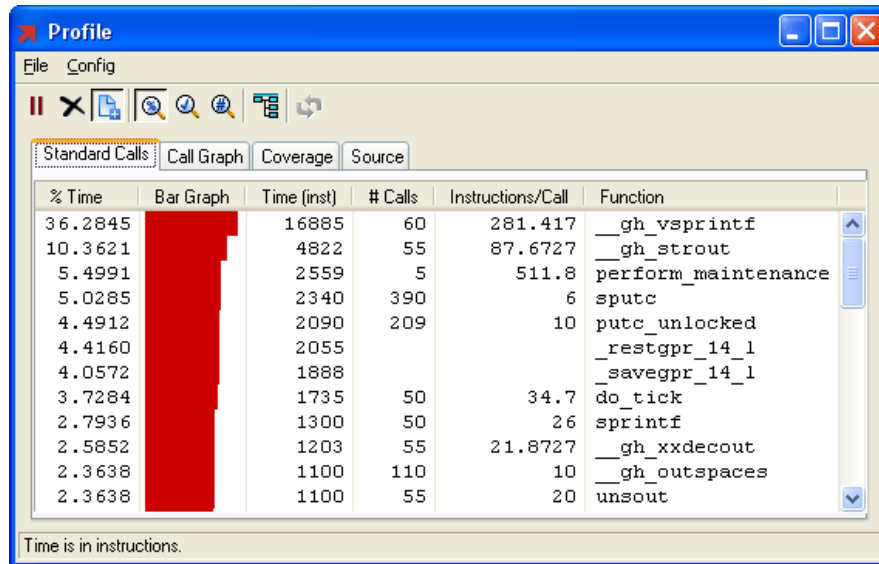
The status report is the report that appears by default when you open the **Profile** window.



This report is available if you are profiling a run-mode task or AddressSpace on an INTEGRITY target, or if you are profiling a stand-alone program. It gives general information about profiling, such as what type of data has been collected. A message in the status bar indicates whether or not profiling data is being collected.

Standard Calls Report

The standard calls report gives a summary of processing time per function. It is available if PC samples are available or if call count data is available. For information about the availability of these data types, see “Overview of Profiling Methods” on page 341.



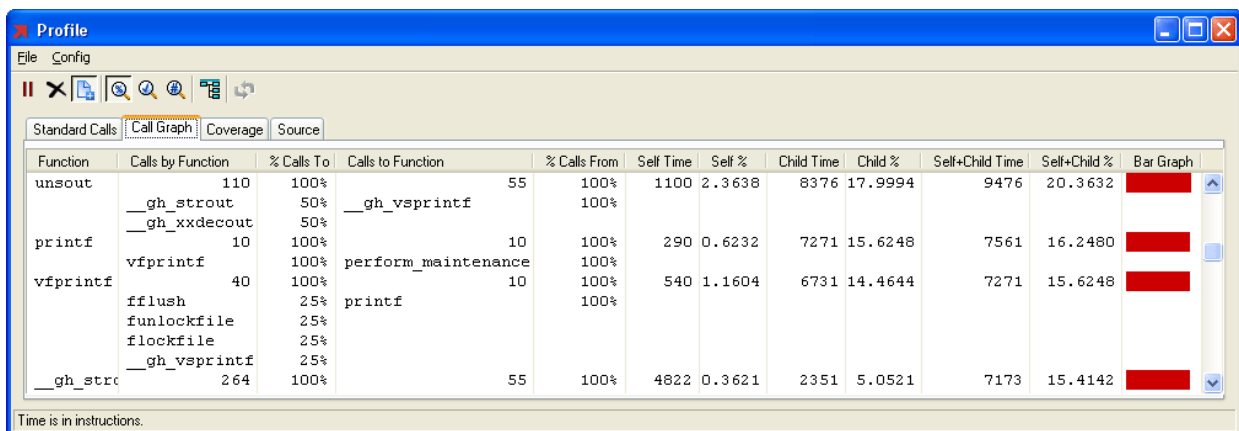
The columns of the standard calls report are described in the following table. Some columns do not contain information unless you used trace data or compiled your program with Builder or driver options that generate call count data. For more information, see “Generating Profiling Data for a Task, AddressSpace, or Stand-Alone Program” on page 342.

Header	Meaning
% Time	The percentage of the total process time spent in each function. (The total of the percentages in this column may not be exactly 100 percent, due to roundoff error.)
Bar Graph	Graphs the value of the % Time column.
Time (unit)	The amount of time in milliseconds (ms), seconds (s), instructions (inst), or cycles (cyc) spent in each function.

Header	Meaning
# Calls	The number of times that the function was called. This column only contains information if call count data is available. For information about the availability of call count data, see “Overview of Profiling Methods” on page 341. If you are profiling a trace-enabled target, call counts are not available for some internal helper functions used by the compiler. Usually these functions have names that begin with one or two underscore characters, and they are called at the beginning and end of many functions to save and restore registers.
Unit/call	The amount of time (in milliseconds, seconds, instructions, or cycles) spent on each call to the function. This column only contains information if call count data is available. For information about the availability of call count data, see “Overview of Profiling Methods” on page 341.
Function	The name of the function.

Call Graph Report

The call graph report gives a summary of processing time spent per function, including time spent in each function’s descendants. (Descendants of a function are all routines directly or indirectly called by that function.) This report is available if call count with call graph profiling data is available. For information about the availability of call count with call graph profiling data, see “Overview of Profiling Methods” on page 341.



Function	Calls by Function	% Calls To	Calls to Function	% Calls From	Self Time	Self %	Child Time	Child %	Self+Child Time	Self+Child %	Bar Graph
unsout	110	100%	55	100%	1100	2.3638	8376	17.9994	9476	20.3632	
__gh_strout	50%		__gh_vsprintf	100%							
__gh_xxdecout	50%										
printf	10	100%	10	100%	290	0.6232	7271	15.6248	7561	16.2480	
vfprintf	100%		perform_maintenance	100%							
fflush	25%		printf	100%							
funlockfile	25%										
flockfile	25%										
__gh_vsprintf	25%										
__gh_strc	264	100%	55	100%	4822	0.3621	2351	5.0521	7173	15.4142	

The columns of the call graph report are described in the following table.

Header	Meaning
Function	The name of the function.
Calls by Function	The functions called by the function or the number of function calls made by the function.
% Calls To	The percent of the total calls made by the function to each of the listed functions.
Calls to Function	The functions that called the function or the number of times that the function was called.
% Calls From	The percent of the total calls made to the function by each of the listed functions.
Self Time	The actual time spent in each function.
Self %	The percentage of total time spent in each function.
Child Time	The actual time spent in all of the children of each function. Child times are calculated by multiplying the amount of time attributed to each child function by the percentage of times the parent called that child.
Child %	The percentage of total time spent in the children of each function.
Self+Child Time	The total processing time spent in each function and its children.
Self+Child %	The percentage of total time spent in each function and its children.
Bar Graph	Graphs the value of the Self+Child % column.

The message in the status bar indicates whether the times are listed in milliseconds, seconds, instructions, or cycles.

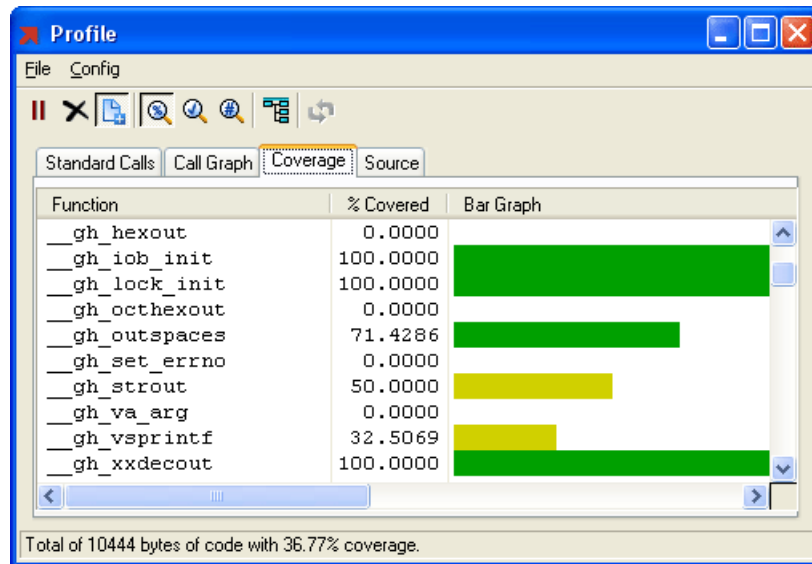
Coverage Report

The coverage report provides the percentage of bytes covered. Examining this report is an easy way to find code that is not being executed. This can be useful for discovering dead code and for verifying that a test suite is testing everything it is intended to test.

Each line in the report provides the percentage of bytes of code that were executed in a function. If profiling data has been generated from trace data, the coverage report includes every function in the AddressSpace or stand-alone program. If coverage analysis data has been generated via compiler instrumentation, the coverage report includes every instrumented function in

the AddressSpace or stand-alone program. Typically, standard library functions are not instrumented. The status bar message gives a summary of coverage for the entire AddressSpace or stand-alone program.

This report is available if coverage analysis data is available. For information about the availability of coverage analysis data, see “Overview of Profiling Methods” on page 341.

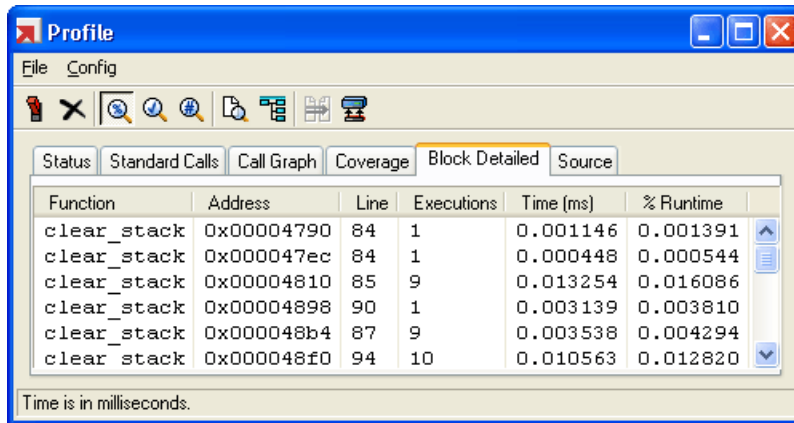


The columns of the coverage report are described in the following table.

Header	Meaning
Function	The function name.
% Covered	The percentage of bytes executed.
Bar Graph	Graphs the value of the % Covered column.

Block Detailed Report

The block detailed report gives the program code coverage per basic block. It is available if coverage analysis data is available and if you did not use trace data. For information about the availability of coverage analysis data, see “Overview of Profiling Methods” on page 341.



The columns of the block detailed report are described in the following table.

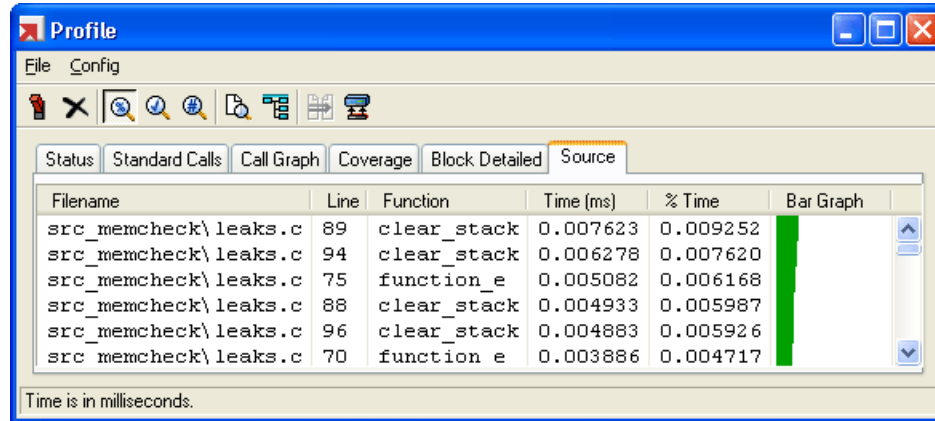
Header	Meaning
Function	The function name.
Address	The starting address of the block.
Line	The file-relative source line corresponding to the block.
Executions	The number of times the block was entered or a NOT REACHED message if zero.
Time (unit)	The total time in milliseconds (ms), seconds (s), or instructions (inst) spent in the block.
% Runtime	The percentage of total time spent in the block.

Source Report

The source report is a list of all the source lines, along with how long each took to execute. A source line is uniquely determined by filename and file-relative line number. Only lines with positive times are displayed. This report is available if PC samples are available. For information about the availability of PC samples, see “Overview of Profiling Methods” on page 341.



Note The time it takes to process this report is proportional to the size of what you are profiling. For example, it takes longer to generate this report for very large programs than for small ones.



The columns of the source report are described in the following table.

Header	Meaning
Filename	The location of the source line.
Line	The file-relative line number of the source line.
Function	The function in which the source line appears.
Time (unit)	The time in milliseconds (ms), seconds (s), instructions (inst), or cycles (cyc) spent on the given line.
% Time	The percentage of the total time spent on the line.
Bar Graph	Graphs the value of the % Time column.

The Profile Window Reference

The following sections provide a comprehensive description of the menu items and buttons that appear in the **Profile** window. For information about the tab views within the **Profile** window, see “Profiling Reports” on page 350.

The File Menu

The following table describes the **File** menu items and lists their command equivalents.

For information about the **profilereport** and **profilemode** commands, see Chapter 13, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.

Menu Item	Meaning	Equivalent Command
Save Report	Saves the text of the report that is currently displayed in the Profile window. The default file extension given to the saved text file is .rep .	profilereport save filename
Append Report	Appends the text of the report currently displayed in the Profile window to a specified, pre-existing file.	profilereport append filename
Print Report	Prints the text of the report currently displayed in the Profile window.	profilereport print
Close	Closes the Profile window, which halts the collection of profiling data and clears existing profiling data.	profilemode close

The Config Menu

The following table describes the **Config** menu items and lists their command equivalents, if any. These menu items are context sensitive; they may not all be available every time you open this menu. Unless otherwise stated, all descriptions of toggle options explain the behavior of the option when enabled.

For information about the **profilemode** command, see Chapter 13, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.

Menu Item	Meaning	Equivalent Command
Config → New Data → Added to Old	Appends new data to old data when a new set of profiling data is processed (such as data from a new run of the program). For more information, see “Adding to or Overwriting Existing Profiling Data” on page 366.	profilemode add
Config → New Data → Replaces Old	Replaces old data with new data when a new set of profiling data is processed (such as data from a new run of the program). For more information, see “Adding to or Overwriting Existing Profiling Data” on page 366.	profilemode replace

Menu Item	Meaning	Equivalent Command
Config → Data Processing → Automatic	Processes and stores profiling data automatically when it is dumped. This is the default unless you are using INTEGRITY. For more information, see “Manually Processing Profiling Data” on page 368.	profilemode automatic
Config → Data Processing → Manual	Does not process profiling data automatically when it is dumped. This is the default if you are using INTEGRITY. For more information, see “Manually Processing Profiling Data” on page 368.	profilemode manual
Config → Function Names → Short	Omits C++ qualifiers from the function names displayed in profiling reports. (You can view fully qualified function names in the tooltips.) This option has no effect on the display of C functions.	profilemode short
Config → Function Names → Long	Displays fully qualified function names in profiling reports. Fully qualified function names include all C++ qualifiers, such as namespace and class names, function arguments, and template information. This option has no effect on the display of C functions.	profilemode long
Config → Time Units → Milliseconds	Displays all times in milliseconds.	profilemode time milliseconds
Config → Time Units → Seconds	Displays all times in seconds.	profilemode time seconds
Config → Time Units → Instructions	Displays all times in instructions.	profilemode time instructions
Config → Time Units → Cycles	Displays all times in cycles.	profilemode time cycles

Menu Item	Meaning	Equivalent Command
Config → Bar Graphs → Linear	Draws all Bar Graph columns in profiling reports as linear graphs.	(no equivalent command)
Config → Bar Graphs → Logarithmic	Draws Bar Graph columns in the Standard Calls , Call Graph , and Source reports as logarithmic graphs. The Bar Graph column in the Coverage report is always drawn as a linear graph regardless of the setting for this option. This is the default.	(no equivalent command)

The Toolbar

The following table describes the buttons available in the **Profile** window and lists their command equivalents, if any. These buttons are context sensitive; different buttons are available in different contexts.

For information about the **profilemode** and **profdump** commands, see Chapter 13, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book. For information about the **browse** command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

Button	Meaning	Equivalent Command
 Stop collecting profile information	Indicates whether profiling data is being collected and, when clicked, toggles profiling to the opposite state (on or off).	profilemode stop
 Start collecting profile information	When this button is in the  position, profiling data is being collected. Clicking the button when it is in this position disables profiling collection and clears any current profiling data (but does not close the Profile window).	
	When this button is in the  position, profiling data is not being collected. Clicking the button when it is in this position enables profiling.	profilemode start

Button	Meaning	Equivalent Command
 Pause updating of profile display	Pauses the continual updating of information in the Profile window. For more information, see “Continual Updates in the Profile Window” on page 349.	no equivalent command
 Clear current profile information	Deletes any existing profiling data. You can use this button in conjunction with the Dump Profiling Info button () to examine profiling data gathered between two points of execution. For more information, see “Manually Dumping Profiling Data” on page 367.	profilemode clear
 Merge new data	Toggles whether new data is added to old data or whether new data replaces old data. When this button appears to be pushed down and when a new set of profiling data is processed (such as data from a new run of the program), new data is added to old data. This is the default. When this button does <i>not</i> appear to be pushed down and when a new set of profiling data is processed, the new data replaces the old data. For more information, see “Adding to or Overwriting Existing Profiling Data” on page 366.	profilemode add and profilemode replace
 Percentage View	Displays, to the left of each source code line in the Debugger, the percentage of time spent in each source line. (In assembly display mode, displays the percentage of time spent on each instruction.) This is the default view in the Debugger. (Note that ~0% means that the particular line/instruction took very close to zero percent of the total execution time.)	profilemode percent
 Coverage View	Highlights dead code (lines that were never executed during the profiling run) in the Debugger.	profilemode coverage

Button	Meaning	Equivalent Command
 Count View	If coverage analysis profiling data is available, clicking this button displays in the Debugger the total number of times each line (or instruction) was executed. If coverage analysis profiling data is unavailable, but call count profiling data is available, clicking this button displays the number of times each function was called. This information is displayed at the beginning of each function in the Debugger. For information about the availability of these data types, see “Overview of Profiling Methods” on page 341.	profilemode count
 Range Analysis	Opens a Range Analysis window. See “Performing Range Analyses” on page 364.	profilemode range start_addr end_addr
 Dynamic Call Graph	Opens a dynamic call graph Browse window centered on the present function being examined in the Debugger. For more information, see “Browsing Dynamic Calls by Function” on page 226.	browse dcalls
 Refresh Report	Updates the report with new data.	no equivalent command
 Process Data	Processes profiling data. For more information, see “Manually Processing Profiling Data” on page 368.	profilemode process
 Dump Profiling Info	Retrieves any currently available profiling data from the target. You can use this button in conjunction with the Clear current profile information button () to examine profiling data gathered between two points of execution. For more information, see “Manually Dumping Profiling Data” on page 367.	profdump
 Close	Closes the Profile window, which halts the collection of profiling data and clears existing profiling data. (Whether this button appears on your toolbar depends on your MULTI configuration.)	profilemode close

Viewing Profiling Information in the Debugger

Once you have collected profiling data, you can use **Profile** window buttons or the **profilemode** command to view some basic profiling information directly in the source pane of the Debugger, as described below. If you profile all tasks in your system, or if you profile a trace-enabled target, the information is continually updated.



Note The following display modes are mutually exclusive (only one type of information can be shown in the Debugger at a time). By default, the Debugger displays time percentage information, if it is available.

- *Time percentage information* — To view the percentage of time spent in each source line, click the **Percentage View** button (📊) in the **Profile** window or run the **profilemode percent** command in the Debugger command pane. The percentage is shown to the left of each source line. (If you are in assembly display mode, the percentage of time spent on each instruction is displayed.) This feature is only available if PC samples or coverage analysis profiling data is available. For information about the availability of these data types, see “Overview of Profiling Methods” on page 341.
- *Coverage information* — To have lines of dead code (lines that were never executed) highlighted in the Debugger, click the **Coverage View** button (📊) in the **Profile** window or run the **profilemode coverage** command in the Debugger command pane. This feature is only available if coverage analysis profiling data is available. For information about the availability of coverage analysis profiling data, see “Overview of Profiling Methods” on page 341.
- *Count information (line executions)* — To view the total number of times each line (or instruction) was executed, click the **Count View** button (📊) in the **Profile** window or run the **profilemode count** command in the Debugger command pane. The number of executions is shown to the left of each source line. This feature is only available if coverage analysis profiling data is available. For information about the availability of coverage analysis profiling data, see “Overview of Profiling Methods” on page 341.
- *Count information (function calls)* — To view the total number of times each function was called, click the **Count View** button (📊) in the **Profile** window or run the **profilemode count** command in the Debugger command pane. The call count for each function is shown to the left of the beginning of each function. This feature is only supported if coverage analysis profiling data is not available and call count profiling data is available.

For information about the availability of these data types, see “Overview of Profiling Methods” on page 341.



Note If there is no profiling data for a particular line, a question mark appears to the left of the line instead of actual count or time percentage information.

For more information about the **profilemode** command, see Chapter 13, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.

If you profile all tasks in your system, profiling information is also displayed in the Debugger’s target list in a column labeled **CPU %**. This column displays the percentage of the CPU that each task and AddressSpace is currently using. For more information, see “Profiling All Tasks in Your System” on page 447.

Performing Range Analyses



Note This information is only relevant if you are profiling a run-mode task or AddressSpace on an INTEGRITY target, or if you are profiling a stand-alone program, *and* if PC samples are available. For information about the availability of PC samples, see “Generating Profiling Data for a Task, AddressSpace, or Stand-Alone Program” on page 342.

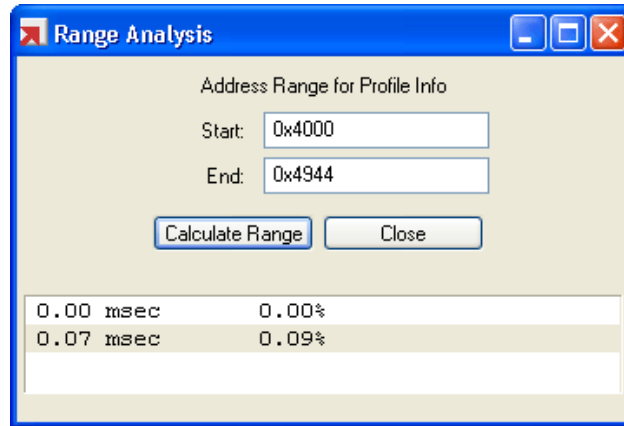
Improving the performance of an application often requires isolating small portions of large functions that account for the majority of the function’s execution time, such as computationally intensive nested loops. MULTI allows you to perform range analyses to collect profiling information for particular sections of code.

You can perform a range analysis from the **Range Analysis** window or from the Debugger command pane. To open a **Range Analysis** window, click the **Range Analysis** button () in the **Profile** window. To specify the range you want to analyze, enter a hexadecimal **Start** and **End** address in the fields of the **Range Analysis** window.



Tip In assembly display mode (see “Source Pane Display Modes” on page 41), hexadecimal addresses are displayed to the left of the corresponding instructions in the Debugger source pane.

After you have specified the range of hexadecimal addresses, click the **Calculate Range** button to display the amount of time (in seconds) that elapsed during execution of the specified range, as well as the percentage of the total execution time required to execute the range.



If no range or an inappropriate range is specified when you click the **Calculate Range** button, the **Range Analysis** window displays an error message.

To perform a range analysis from the Debugger command pane, issue the following command:

```
profilemode range start_addr end_addr
```

where *start_addr* and *end_addr* are the beginning and end of your range, respectively. The result of the analysis appears in the Debugger command pane.

Managing Profiling Data

You can manage collected profiling data in a number of ways: by choosing whether or not to overwrite existing profiling data with new data, by manually dumping and processing your profiling data, and by clearing existing data. Support for some of these capabilities depends on what you are profiling and on the method of profiling. The following sections provide more information.

Adding to or Overwriting Existing Profiling Data

When new profiling data is collected and processed, it can be added to profiling data that already exists, or it can replace old profiling data.

Accumulating data across multiple executions may be useful for getting a more accurate understanding of application performance. This is the default behavior (with one exception*). To append new profiling data to existing profiling data:

- If you are profiling all tasks in your system or if you are profiling a trace-enabled target — In the **Profile** window, click the **Merge new data** button () so that it appears to be pushed down, or enter **profilemode add** in the Debugger command pane.
- If you are profiling a task, AddressSpace, or stand-alone program — Select **Config → New Data → Added to Old** in the **Profile** window, or enter **profilemode add** in the Debugger command pane. (*Note that you cannot add new data to existing data if you are profiling a task or AddressSpace, are using INTEGRITY version 10 or later, and have established a run-mode connection.)

To replace existing profiling data with new profiling data:

- If you are profiling all tasks in your system or if you are profiling a trace-enabled target — In the **Profile** window, click the **Merge new data** button () so that it does *not* appear to be pushed down, or enter **profilemode replace** in the Debugger command pane.
- If you are profiling a task, AddressSpace, or stand-alone program — Select **Config → New Data → Replaces Old** in the **Profile** window, or enter **profilemode replace** in the Debugger command pane.

For information about the **profilemode** command, see Chapter 13, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.

Manually Dumping Profiling Data



Note This information is only relevant if one of the following is true:

- You are profiling a run-mode task or AddressSpace on an INTEGRITY target, or you are profiling a stand-alone program, *and* you instrumented your code to generate profiling data (that is, you compiled your program with Builder or driver profiling options). For more information, see “Generating Profiling Data for a Task, AddressSpace, or Stand-Alone Program” on page 342.
- You collected PC samples over a freeze-mode connection.

By default, MULTI processes collected profiling data automatically each time your program exits. In some situations, however, you may want to halt your process and dump the collected profiling data at some other point. The most common scenarios for manually dumping profiling data are:

- When you are debugging a process, such as an operating system, that does not terminate.
- When you want to dump profiling data gathered between two points of execution so that you can profile specific areas of execution.



Note MULTI’s ability to dump profiling data depends on command line procedure calls. If you are not using INTEGRITY, interrupts may need to be disabled before command line procedure calls and, consequently, dumps will work. If you need to disable interrupts on your target, first halt your target, and then disable interrupts. After dumping profiling data (explained next), turn interrupts back on and run the program.

To dump all available forms of profiling data (PC samples, call count data, and coverage analysis data), halt your process and then do one of the following:

- Click the **Dump Profiling Info** button (.
- In the Debugger command pane, enter the **profdump** command. For information about the **profdump** command, see Chapter 13, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.

To dump profiling data gathered between two specific points of execution:

1. Run the program to the first point of execution and then clear any profiling data gathered prior to reaching that point. To clear profiling data, do one of the following:
 - Click the **Clear current profile information** button (✕).
 - In the Debugger command pane, enter the **profilemode clear** command. For information about the **profilemode** command, see Chapter 13, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.
2. Run the program to the second point of execution and then dump the profiling data gathered between the two points.

If automatic data processing is disabled, you must manually process dumped profiling data. For more information, see “Manually Processing Profiling Data” on page 368.

For information about the limitations associated with dumping profiling data, see “Caveats for Dumping and Clearing Profiling Data” on page 370.

Manually Processing Profiling Data



Note This information is only relevant if you are profiling a run-mode task or AddressSpace on an INTEGRITY target, or if you are profiling a stand-alone program.

Unless you are using INTEGRITY, MULTI is set by default to automatically process and store profiling data when it is dumped. If you want more control over when profiling data is processed, you can disable automatic processing and process your profiling data manually.

To disable the automatic processing of profiling data, do one of the following:

- In the **Profile** window, select **Config** → **Data Processing** → **Manual**.
- In the Debugger command pane, enter the **profilemode manual** command.

After you have disabled automatic processing, you will have to process your profiling data manually. To successfully process profiling data (automatically or manually), you should be connected to your target with the program loaded.



Note To avoid inadvertently merging old data into a profiling report, MULTI deletes profiling data files after processing them. To save these files, copy the *.out files to a separate directory before manually processing profiling data.

To start the manual processing of data, do one of the following:

- Click the **Process Data** button ()
- In the Debugger command pane, enter the **profilemode process** command.

To load a .pro file output by **protrans** (see “Using the protrans Utility” on page 371), perform the following steps:

1. Make sure you are connected to your target and the program is loaded.
2. Copy saved profiling data files to your run directory.
3. Rename *.pro files in the format *program.pro*, where *program* is the name of your program.
4. Open the **Profile** window (for instructions, see “Collecting Profiling Data for a Task, AddressSpace, or Stand-Alone Program” on page 345).
5. Enter the **profilemode import** command in the Debugger command pane.
6. Follow the preceding instructions for processing data manually.

To re-enable automatic processing of profiling data, do one of the following:

- Choose **Config → Data Processing → Automatic**.
- In the Debugger command pane, enter the **profilemode automatic** command.

For information about the **profilemode** command, see Chapter 13, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.

Clearing Profiling Data

To clear existing profiling data, do one of the following:

- In the **Profile** window, click the **Clear current profile information** button (X).
- In the Debugger command pane, enter the **profilemode clear** command. For information about the **profilemode** command, see Chapter 13, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.
- In the **Profile** window, click the **Close** button (X) to close the **Profile** window, halt the collection of profiling data, and clear existing profiling data. (Whether this button appears on your toolbar depends on your MULTI configuration.)

For information about the limitations associated with clearing profiling data, see “Caveats for Dumping and Clearing Profiling Data” on page 370.

Caveats for Dumping and Clearing Profiling Data

- MULTI uses command line procedure calls to dump and clear profiling data. In some cases command line procedure calls may not be possible or may have adverse effects. If you are using INTEGRITY, command line procedure calls are unsafe for tasks that have been halted while performing a system call. MULTI attempts to detect this situation and prevent command line procedure calls in this context, but it is not always able to do so, nor is it able to detect every situation in which it is unsafe to make command line procedure calls. To guarantee correct operation, you must ensure that the task you are profiling is halted in a safe location prior to dumping or clearing profiling data. For more information about the limitations of command line procedure calls, see “Caveats for Command Line Procedure Calls” on page 286.
- If you are not using INTEGRITY, interrupts may need to be disabled before command line procedure calls and, consequently, dump or clear operations will work. If you need to disable interrupts on your target, first halt your target, and then disable interrupts. After dumping or clearing profiling data, turn interrupts back on and run the program.
- If you profile a run-mode task or AddressSpace on an INTEGRITY target, or if you profile a stand-alone program, *and* if you instrumented your code to generate profiling data, dumping or clearing the profiling data may cause

the program, task, or tasks inside the AddressSpace to run for the duration of the dump or clear operation. This brief run may cause MULTI to generate PC samples.

Using the protrans Utility



Note This information is only relevant if you are profiling a run-mode task or AddressSpace on an INTEGRITY target, or if you are profiling a stand-alone program.

Running **protrans** separately is useful when you want to automate the acquisition of large amounts of profiling data and then use the **Profile** window to display the data once all the executions are complete. Normally, when you use MULTI to collect profiling data, the actions of the **protrans** utility are transparent to you. This utility reads the profiling data that is produced when a program is executed, and it can accumulate profiling data over multiple executions of a program. It then translates the data into an intermediate format that the **Profile** window uses when displaying profiling information.

The information in this section is provided for advanced users who want to run **protrans** outside of the MULTI environment. To run this utility manually, issue the following command from a command shell:

```
protrans options program
```

where *options* can be any non-conflicting combination of the options described in the following table.

Option	Meaning
-a	Adds the profiling data for the current execution to a summary profile with a .pro extension. Without this switch, a new profile, which overwrites the previous profile, is generated with each execution.
-e	Writes the error messages to a file with the same base name and an .err extension.
-q	Suppresses message printing. Without this switch, protrans prints a small message for each execution, describing the type of profiling data found.

Option	Meaning
-m file	Specifies <i>file</i> as a mon.out file containing call count (but not call graph) profiling data. You can specify multiple files with multiple uses of this option.
-g file	Specifies <i>file</i> as a gmon.out file containing call count with call graph profiling data. You can specify multiple files with multiple uses of this option.
-b file	Specifies <i>file</i> as a bmon.out file containing coverage analysis profiling data. You can specify multiple files with multiple uses of this option.
-d file	Specifies <i>file</i> as a .dbp file containing PC samples. The contents of the file depend on the data collected.
-B	Specifies the target is big endian. The default endianness is that of the host.
-L	Specifies the target is little endian. The default endianness is that of the host.

Suppose you have a big endian PowerPC program called **dylan**, built with call graph and coverage analysis profiling, and with a collection of sample input files: **sampleinput1**, **sampleinput2**, etc. Now consider the following **cs**h shell script for UNIX hosts.

```
#!/bin/csh
foreach p (sampleinput*)
    simppc dylan $p
    protrans -a -B -q dylan
end
rm -f gmon.out bmon.out
```

This script repeatedly runs the **dylan** program with the sample input using **simppc**, and calls the **protrans** utility to read in the generated profiling data for each execution. The argument **dylan** specifies the profiled program for the generated data (the default is **a.out**).

After the script finishes, an intermediate profiling data file is generated that contains the summary profile. The intermediate file has a **.pro** extension appended to the program name if it has no extension.

In the preceding example, the file **dylan.pro** is generated in the same directory where the shell script is executed.

After you have generated this file, you can process the data in the Debugger (see “Manually Processing Profiling Data” on page 368). You can read in the summary profile stored in **dylan.pro**, and then use the various features of the **Profile** window to view the information.

You can also dump the contents of this file using the **gdump** utility, which will display the data in an easy-to-read format. For information about the **gdump** utility, see the *MULTI: Building Applications* book for your target processor family.

The last line in the preceding script deletes any remaining profiling data files after executions of the profiled program. By default, the **protrans** utility looks for the following files:

- **mon.out** — This file is produced after running a program built for call count profiling (without call graph support).
- **gmon.out** — This file is produced after running a program built for call count profiling with call graph support enabled.



Note A program can only produce one or the other of **mon.out** or **gmon.out**.

- **bmon.out** — This file is produced after running a program built for coverage analysis. This type of profiling is done alone or in conjunction with either call count profiling or call count with call graph profiling.

For more information about the profiling options that generate these files, see “Generating Profiling Data for a Task, AddressSpace, or Stand-Alone Program” on page 342.

You can also specify other specific data files to **protrans**.

Thus, the shell script example for UNIX hosts that is given earlier could be rewritten to be:

```
#!/bin/csh
foreach p (sampleinput*)
    simppc dylan $p
    mv gmon.out gmon.$p
    mv bmon.out bmon.$p
end
foreach p (sampleinput*)
    protrans -a -B -q -g gmon.$p -b bmon.$p dylan
    rm -f gmon.$p bmon.$p
end
```

The first loop runs **dylan** with the sample input and stores the generated data files into uniquely named temporary files. The second loop then calls **protrans** to read in the data from these files and produces a summary profile. The **-g** option specifies a call count with call graph profiling data file, and the **-b** option specifies a coverage analysis data file. You can specify multiple files of a single profiling data type with multiple uses of these options. For example:

```
protrans -g gmon.1 -g gmon.2 -g gmon.3
```


Using Other View Windows

Chapter 16

This Chapter Contains:

- Viewing Call Stacks
- Viewing Native Processes
- Viewing Caches

Viewing Call Stacks

The **Call Stack** window displays call stacks (also known as *call stack traces* or simply *stack traces*), which consist of stack frames that are currently active in your program. Each stack frame typically represents a function call. Stack frames are shown in order from most to least recently created.

To open the **Call Stack** window, do one of the following:

- Click the **Call Stack** button () .
- Choose **View → Call Stack**.
- In the command pane, enter **callsviiew**. For information about the **callsviiew** command, see Chapter 6, “Call Stack Command Reference” in the *MULTI: Debugging Command Reference* book.

The following table describes the buttons available from the toolbar of the **Call Stack** window.

Button	Effect
	Toggles the display of parameters in function calls.
	Toggles the display of the filename and line number of the function call.
	Toggles whether to display stack frames before <code>main()</code> .
	Toggles whether the window is refreshed.
	Opens an Editor on the selected function, if it has source code.
	Opens a Data Explorer to show all the local variables of the selected function.
	Prints the text contents of the Call Stack window to your printer. To print call stack information to the command pane, enter the calls command. This command is useful if you want to copy the resulting text and paste it into an email or another document. For information about the calls command, see Chapter 6, “Call Stack Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
	Closes the Call Stack window. This button may not appear if you have configured MULTI to hide the close buttons.

At the far right of the toolbar is the **Max Depth** field, which controls the maximum number of stack levels which the window will display. You can

change it according to your preference. For example, if you are debugging a program on a very slow target and you only want information about the first few levels of the call stack, you can decrease the number so that the window is refreshed more quickly.

Below the toolbar is the call stack pane, where the call stack is displayed. The following table lists the mouse and keyboard operations you can perform in the call stack pane.

To	Do this
Display a function in the source pane	Click the function
Open an Editor on a function	Double-click the function
Copy the entire call stack pane to the clipboard	Press Ctrl + Shift + C
Search forward in the call stack pane	Press Ctrl + F
Search backward in the call stack pane	Press Ctrl + B

During a debugging session, if you change the attributes of a **Call Stack** window, the changes will affect subsequently created **Call Stack** windows. You can also change these attributes with the **cvconfig** command. For information about this command, see Chapter 6, “Call Stack Command Reference” in the *MULTI: Debugging Command Reference* book.

The Call Stack Window and Command Line Procedure Calls

If a breakpoint is hit during the execution of a command line procedure call, the call stack pane of the **Call Stack** window displays a separator line corresponding to the location where the target was stopped when you made the procedure call. The lower portion is the call stack from before the function call; the upper portion is the call stack starting from the command line procedure call.

The Call Stack Window and Procedure Prologues and Epilogues

At the beginning and end of every procedure are special code sequences called the prologue and epilogue, respectively. These special code sequences save and restore registers and modify the stack pointer. Full source-level debugging is not possible within these regions, so MULTI does not display source-level breakpoints for these lines. You can single-step through this code at the machine level, but many other tasks such as tracing the stack and examining variables are not supported until you are outside this region. The **Call Stack** window may

display incorrect information when a process is stopped inside a prologue, an epilogue, or a function called by a prologue or epilogue.

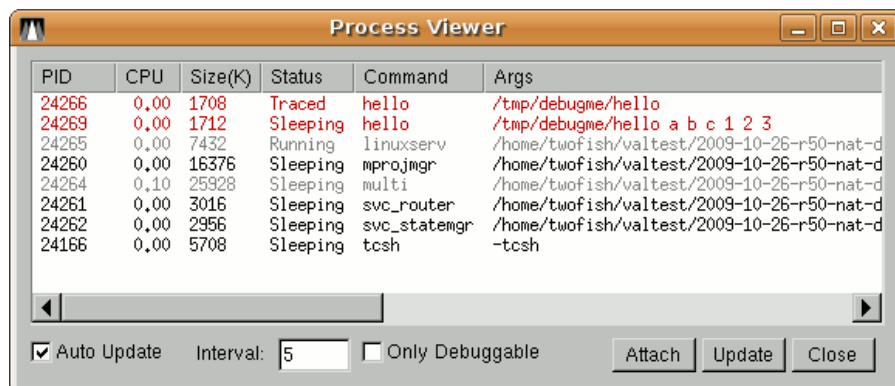
Viewing Native Processes

The **Process Viewer** displays a snapshot of the processes on your native Linux or Solaris target, much like the UNIX `top` and `ps` utilities. The primary purpose of this window is to allow you to attach to native processes. The **Process Viewer** also provides an easy way to identify PIDs for use with certain Debugger commands or dialog boxes.

To open a **Process Viewer**, do one of the following from your native debugging environment:

- Select **View → Task Manager** from the Debugger menu bar. (If you are in a non-native environment and in run mode, this menu selection will open a Task Manager instead of a **Process Viewer**.)
- Issue the **top** command from the Debugger command pane. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.
- Issue the command **multi -top** from your shell (see Appendix C).

A sample **Process Viewer** for Linux is displayed below:



PID	CPU	Size(K)	Status	Command	Args
24266	0.00	1708	Traced	hello	/tmp/debugme/hello
24269	0.00	1712	Sleeping	hello	/tmp/debugme/hello a b c 1 2 3
24265	0.00	7432	Running	linuxserv	/home/twofish/valtest/2009-10-26-r50-nat-d
24260	0.00	16376	Sleeping	mprojmgr	/home/twofish/valtest/2009-10-26-r50-nat-d
24264	0.10	25928	Sleeping	multi	/home/twofish/valtest/2009-10-26-r50-nat-d
24261	0.00	3016	Sleeping	svc_router	/home/twofish/valtest/2009-10-26-r50-nat-d
24262	0.00	2956	Sleeping	svc_statemgr	/home/twofish/valtest/2009-10-26-r50-nat-d
24166	0.00	5708	Sleeping	tcsh	-tcsh

The **Process Viewer** displays all of the processes that you own, color-coded depending on their type, where:

- Red — Indicates a process that carries debugging information and contains source code that can be debugged.

- Purple — Indicates a process that has a dash as the first character in `argv[0]`. This typically means that the process is a debug server launched by the MULTI Debugger and should not be debugged.
- Gray — Indicates a process or child process of the MULTI Debugger. Attaching to this process will cause MULTI to stop functioning correctly.
- Black — Indicates a process that does not have available debugging information.

To attach to a process and debug it, either select the process and click **Attach**, or simply double-click the process. If the Debugger cannot locate the executable image associated with the process, you will be prompted to choose it from the disk.

If the **Process Viewer**'s contents become out of date and you want to see the current list of processes available on your native target, simply click **Update** to refresh the list. You can also enable automatic periodic refreshing by selecting the **Auto Update** check box and entering an integer in the **Interval** text box, which specifies how often (in seconds) updates should occur. The default interval is 5 seconds.

Viewing Caches

MULTI includes two display windows that allow you to view the contents of the caches on your processor. You can use the **Cache View** and **Cache Find** windows to browse cache contents and search all of the caches at a specific address.

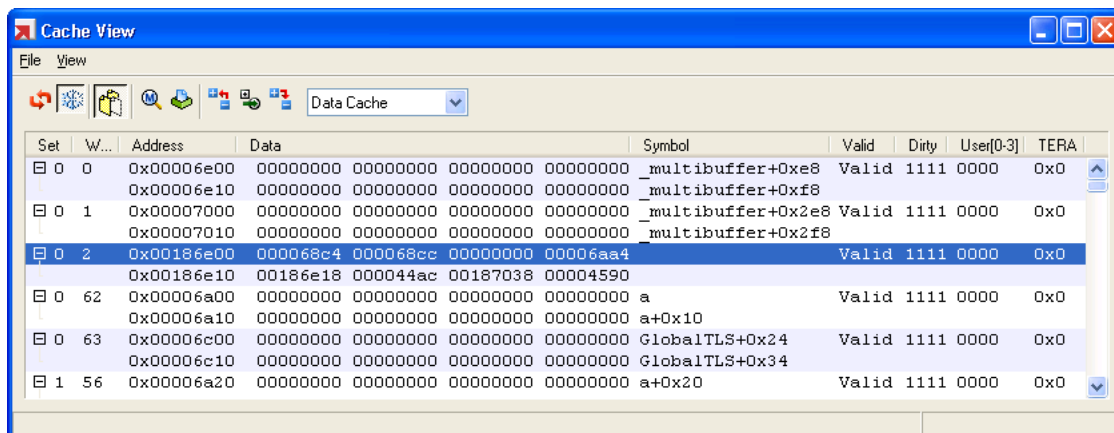


Note Support for cache viewing and searching varies according to processor type. Not all processors support these features. Availability depends on your debug server connection. For information about whether cache viewing is supported, see the *MULTI: Configuring Connections* book for your target processor.

The Cache View Window

The **Cache View** window displays all of the entries from a single cache. You can use this window to browse through the cache data to see what addresses and data are in the cache, what addresses are in the cache but have been invalidated, and other information. To open a **Cache View** window, enter the

cacheview command in the Debugger command pane. (For information about this command, see “Cache View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.) The **Cache View** window is shown next.



The drop-down box on the toolbar allows you to select which cache to view. The list that appears when you click the arrow in this box will display the caches available for viewing, which will vary according to your processor type. The possible choices are: **Data Cache**, **Instruction Cache**, **Unified L1 Cache**, **L2 Cache** and **L3 Cache**. By default, the **Cache View** window hides invalid cache entries. You can change this default behavior by clicking the  button.

The columns in the **Cache View** window also vary depending on your processor and cache type. The following table describes the columns that are displayed for all processor types. For information about the additional columns that are specific to your processor, see “Processor-Specific Columns” on page 382.

Column	Description
Set	Displays the set of the cache line. Each address maps to a single set in the cache.
Way	Displays the way of the cache line. Any address that maps to the current set can be found in any way within that set. This column does not apply for direct-mapped caches.
Address	Displays the address of the cache line. This address may be a virtual or physical address, depending on your processor, cache and whether the memory management unit is enabled.

Column	Description
Data	Displays the data associated with this cache line. The number of bytes displayed depends on your processor and cache type.
Symbol	Displays the symbol associated with the address at the start of the cache line.

You can sort the data in the **Cache View** window by clicking the column header above the column that you want to sort by. You can also change the display, refresh the window, or open other windows using the buttons described in the following table.

Button	Effect
	Refreshes the current view by reloading cache data from the target.
	Hides or shows all invalid cache entries. When the button is pushed down, invalid caches entries are not displayed. Otherwise, all cache entries, including invalid entries, are visible.
	Opens a Memory View window at the address of the selected cache line.
	Opens a Cache Find window at the address of the selected cache line. See “The Cache Find Window” on page 383.
	Contracts all entries in the Cache View window so that each cache line takes up a single line in the cache list.
	Expands all valid entries and contracts all invalid entries in the Cache View window so that each valid cache line is fully displayed and each invalid entry takes up a single line.
	Expands all entries in the Cache View window so that each cache line is fully displayed.

Processor-Specific Columns

The columns that will appear in the **Cache View** window for the supported processors are listed in the following table. See your processor's manual for more information about the data in the caches.



Note This list of processors may not be comprehensive. If your processor type does not appear, contact Green Hills Software to see if cache viewing is supported for your processor type.

Processor	Cache	Columns
PowerPC 440	Data Cache	Valid — Indicates whether the cache line is valid. Dirty — Indicates whether the cache line is dirty. Each of the 4 bits corresponds to a double word of data in the cache line. User — Displays the 4 user bits associated with this cache entry. TERA — Displays the tag extended real address (TERA) of this cache entry. This is the top 4 bits of the physical address associated with this cache entry.
PowerPC 440	Instruction Cache	Valid — Indicates whether the cache line is valid. TS — Displays the translation space for the memory page associated with this cache line. TD — Displays the TID disable field for the memory page associated with this cache entry. TID — Displays the translation ID field for the memory page associated with this cache entry.
PowerPC 8260	Data Cache	Valid — Indicates whether the cache line is valid. Dirty — Indicates whether the cache line is dirty.
PowerPC 8260	Instruction Cache	Valid — Indicates whether the cache line is valid.
PowerPC 7447, PowerPC 7450	Data Cache	Valid — Indicates whether the cache line is valid. Dirty — Indicates whether the cache line is dirty. Shared — Indicates whether the cache line is shared in a multiprocessor system.

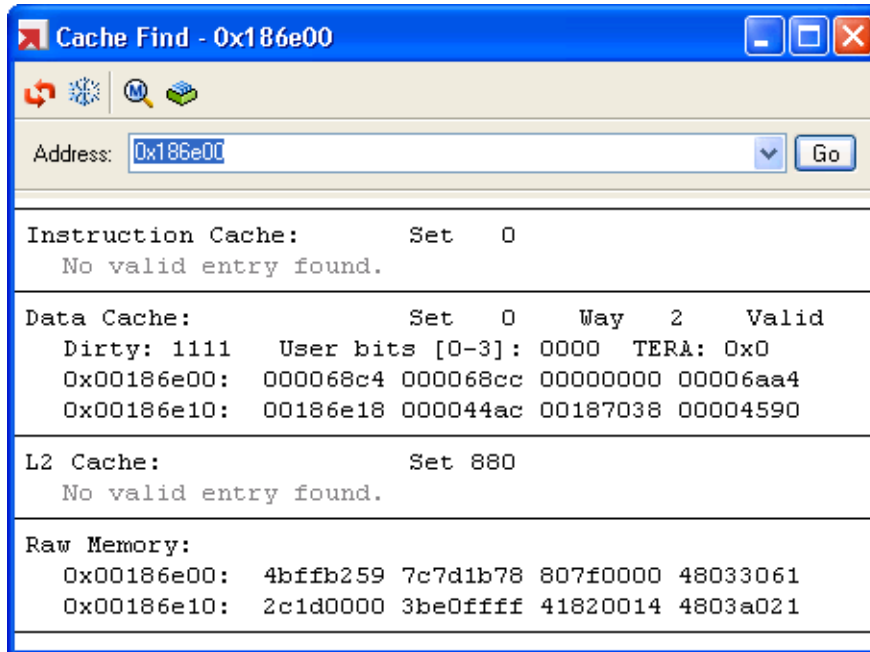
Processor	Cache	Columns
PowerPC 7447, PowerPC 7450	Instruction Cache	Valid — Indicates whether the cache line is valid.
PowerPC 7447, PowerPC 7450	L2 Cache	A0 — Displays the allocate bit for the first cache block of this cache entry. MESI0 — Displays the <code>MESI</code> bits for the first cache block of this cache entry A1 — Displays the allocate bit for the second cache block of this cache entry. MESI1 — Displays the <code>MESI</code> bits for the second cache block of this cache entry
PowerPC 7450	L3 Cache	S[X]:MESI — Displays the <code>MESI</code> bits for cache block <code>x</code> of this cache entry. S[X]:Alloc — Displays the allocate bit for cache block <code>x</code> . SX:Parity — Displays the parity bit for cache block <code>x</code> .

The Cache Find Window

The **Cache Find** window allows you to view the contents of all of your processor's caches, as well as the memory underneath the caches, at a specific address. To open a **Cache Find** window, do one of the following:

- In the Debugger command pane, enter the **cachefind** command. For information about this command, see “Cache View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.
- In the **Cache View** window, click .

The **Cache Find** window is shown next.



The **Cache Find** window displays the valid line that contains the specified address for each cache. If the address is not found in a cache, the set that was searched will be displayed and the window will indicate that the address was not found in that cache. If a cache line is found that contains the address, the cache tags associated with the cache line are displayed along with the data from the cache line.

The data in the **Cache Find** window is colored to indicate the state of the data. This makes the display easy to read and can help you to find cache coherency problems. A cache coherency problem occurs when there is data in the cache that is clean (not dirty) and it differs from data in memory or a cache further from the processor. In this case, the data that makes up the cache coherency conflict is colored in red. In addition, data in memory that has a dirty cache line associated with it is colored gray.

To refresh the cache data in the **Cache Find** window, click . To change the address to search for, enter a new address in the **Address** field and click **Go on Selected Items**.

To freeze the cache data in the **Cache Find** window so that it does not update when the target changes state, click . (The window is frozen when this button appears pushed down. Click the button again to unfreeze the window.)

To open a **Memory View** window at the specified address, click . (See Chapter 13, “Using the Memory View Window” for information about using this window.)

To open a **Cache View** window, click .

Advanced Debugging in Specific Environments

Part III

Testing Target Memory

Chapter 17

This Chapter Contains:

- Quick Memory Testing: Using the Memory Test Wizard
- Advanced Memory Testing: Using the Perform Memory Test Window
- Viewing Memory Test Results
- Continuous Memory Testing
- Memory Testing with a Target Agent
- Types of Memory Tests
- Efficient Testing Methods
- Running Memory Tests from the Command Line
- Detecting Coherency Errors

MULTI allows you to perform a number of destructive and nondestructive tests on your target's memory. Destructive tests overwrite the contents of memory in the test region. Two graphical tools, the **Memory Test Wizard** and the **Perform Memory Test** window, provide a simple way to configure and perform memory tests.

Additionally, MULTI offers manual and optional automatic methods for checking coherency between target memory and an original executable program file. These tools and available types of memory testing are described in this chapter.



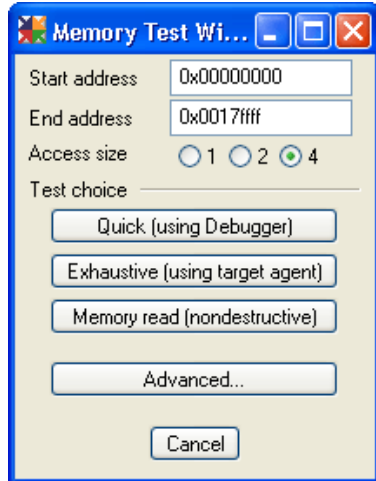
Note You can perform all of the memory tests by issuing the **memtest** command in the Debugger command pane. However, the **Perform Memory Test** window provides a simpler way to control all of the same behaviors and parameters that you can configure with the more complicated **memtest** command. For the complete syntax of the **memtest** command, see “Running Memory Tests from the Command Line” on page 414.

Quick Memory Testing: Using the Memory Test Wizard

The **Memory Test Wizard** allows you to configure and launch the most common memory tests quickly. If you want to set options or run tests other than those listed on the **Memory Test Wizard**, click **Advanced** to open the **Perform Memory Test** window. For more information, see “Advanced Memory Testing: Using the Perform Memory Test Window” on page 394.

To launch the **Memory Test Wizard**, connect to your target, and select **Target** → **Memory Test** from the Debugger.

A sample **Memory Test Wizard** is shown next.



This window allows you to set the basic parameters for your memory test (i.e., the start address and end address of the area of memory to be tested, and the access size to be used while performing the test) and quickly launch one of three common test combinations.

To configure your memory test from the **Memory Test Wizard**, first use the fields described next to define the memory area and access size for the test.

Start address	Defines the lowest address to test. Enter a valid address in this field. The Memory Test Wizard interprets the value as a signed 32-bit integer, so you may only enter addresses between 0 and 0x7fffffff. (Note that the memtest command supports addresses up to 0xffffffff. For information about the memtest command, see “Running Memory Tests from the Command Line” on page 414.) This field defaults to 0x00000000.
End address	Defines the highest address to test. Enter a valid address in this field. This address must be greater than the start address. The Memory Test Wizard interprets the value as a signed 32-bit integer, so you may only enter addresses between 0 and 0x7fffffff. (Note that the memtest command supports addresses up to 0xffffffff. For information about the memtest command, see “Running Memory Tests from the Command Line” on page 414.) This field defaults to 0x00000000. You can use the Debugger to compute the end address if you know the memory area’s start address and size. For example, if your memory area begins at 0x10800000 and is 256 KB in size, the following command will print the end address: <pre>> print/x 0x10800000 + 256*1024 - 1 0x1083ffff</pre>
Access size	Specifies the access size, in bytes, to be used while performing the memory test. Select 1, 2, or 4. The default is 4.



Note Valid values that you enter in the **Start address**, **End address**, and **Access size** fields are carried over to the **Perform Memory Test** window if you select **Advanced**.

The **Test choice** section of the **Memory Test Wizard** allows you to select one of the tests or test combinations listed in the following table.

Quick (using Debugger)	Performs the address bus walking test and data bus walking test, with both walking ones and zeros. When launched with this button, these tests are performed by the Debugger. See “Address Bus Walking Test” on page 404 and “Data Bus Walking Test” on page 406 for a full description of these tests.
Exhaustive (using target agent)	Performs the address bus walking test and data bus walking test, with both walking ones and zeros, as well as the data pattern test. The pattern test uses the pseudorandom pattern and the maximize address bus transition options (see “Data Pattern Test” on page 408 for details). When launched with this button, these tests are performed using a target agent. The target agent is not available for all targets. If no target agent is available for your target environment, this button will be dimmed. For more information about target agents, see “Memory Testing with a Target Agent” on page 403. This test uses the target agent positioned at the start of the test range. See “Address Bus Walking Test” on page 404, “Data Bus Walking Test” on page 406, and “Data Pattern Test” on page 408 for a full description of these tests.
Memory read (nondestructive)	Performs the memory read test. When launched with this button, this test is performed by the Debugger. See “Memory Read Test” on page 411 for a full description of this test.

Clicking one of the preceding buttons launches the corresponding memory test(s) on the target that was selected in the target list when you opened the **Memory Test Wizard**. A **Memory Test Results** window opens (see “Viewing Memory Test Results” on page 402) and remains open after the test completes or is aborted.



Note To abort memory testing, press the Esc key.

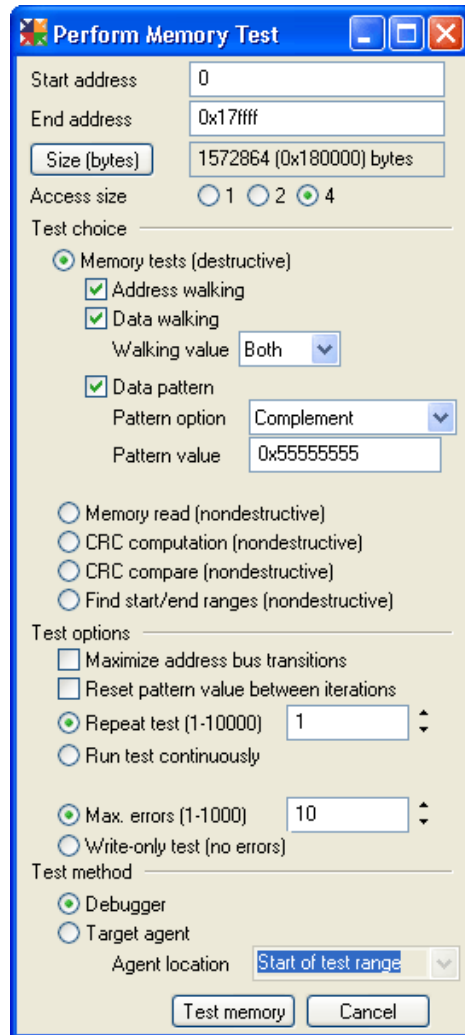
Advanced Memory Testing: Using the Perform Memory Test Window

The **Perform Memory Test** window provides more options and control for configuring memory tests than the **Memory Test Wizard**.

To open the **Perform Memory Test** window:

1. Connect to your target.
2. Select **Target** → **Memory Test** from the Debugger window.
3. Click **Advanced** in the **Memory Test Wizard** that appears.

A sample **Perform Memory Test** window is pictured below. Valid values specified in the **Memory Test Wizard** are carried over to the **Perform Memory Test** window.



This window allows you to specify the area of memory to be tested, the type of memory tests to be performed, and various options about how those tests are performed. The following sections explain how to configure all of these settings.

Defining Memory Areas to Test

The fields at the top section of the **Perform Memory Test** allow you to define the memory area and access size for your memory test.

Start address	0
End address	0x17ffff
Size (bytes)	1572864 (0x180000) bytes
Access size	☐ 1 ☐ 2 ☒ 4

These fields are described in the following table.



Note Three of these fields are identical to fields in the **Memory Test Wizard**. Any valid values you may have input in these fields in the wizard are carried over to the **Perform Memory Test** window.

Start address	Defines the lowest address to test. This field defaults to 0, or to the value entered in the Memory Test Wizard , if any. For more information, see the description of Start address in “Quick Memory Testing: Using the Memory Test Wizard” on page 390.
End address	Defines the highest address to test. This field defaults to 0, or to the value entered in the Memory Test Wizard , if any. For more information, see the description of End address in “Quick Memory Testing: Using the Memory Test Wizard” on page 390.
Size (bytes)	Displays the total size of the area of memory to be tested. This field is provided for your information; the memory test itself uses the specified Start address and End address values. Click the Size (bytes) button if you want to calculate or recalculate the size after changing the Start address or End address .
Access size	Specifies the access size, in bytes, to be used while performing the memory test. Select 1, 2, or 4. This setting defaults to 4, or to the choice entered in the Memory Test Wizard , if any.

Selecting Memory Tests

The second section of the **Perform Memory Test** window allows you to select which memory test(s) to perform.

Test choice

☒ Memory tests (destructive)

☒ Address walking

☒ Data walking

Walking value: Both

☒ Data pattern

Pattern option: Complement

Pattern value: 0x55555555

☐ Memory read (nondestructive)

☐ CRC computation (nondestructive)

☐ CRC compare (nondestructive)

☐ Find start/end ranges (nondestructive)

The test choices include both *destructive* and *nondestructive* tests. Destructive tests overwrite the contents of memory in the test region. You can run more than one type of destructive test simultaneously, but nondestructive tests must be run individually. The following table briefly describes the test choices. Each test is described in more detail in “Types of Memory Tests” on page 404.

Address walking	Selects the address bus walking test. This destructive memory test verifies that address lines are not stuck at one or zero and are not tied together. For more details about this test, see “Address Bus Walking Test” on page 404. This test can be run at the same time as other destructive memory tests.
Data walking	Selects the data bus walking test. This destructive memory test verifies that data bus lines are not stuck at one or zero and are not tied together. For more details about this test, see “Data Bus Walking Test” on page 406. This test can be run at the same time as other destructive memory tests.

Walking value	Determines the walking value for address walking and/or data walking tests. Make a selection to specify whether these tests should be performed with a walking value that is a single one bit surrounded by zero bits (One) or a single zero bit surrounded by one bits (Zero), or if walking tests should be performed twice, once with the walking value <i>one</i> and once with the walking value <i>zero</i> (Both). The default setting is Both. This option is only available if you have selected the address walking and/or data walking test(s).
Data pattern	Selects the data pattern test. This destructive memory test verifies that all memory addresses in the range can successfully store values. For more details about this test, see “Data Pattern Test” on page 408. This test can be run at the same time as other destructive memory tests.
Pattern option	Determines what type of modifications will be performed on the data pattern between iterations when performing a data pattern test. Select Static, Complement, Rotate, Rotate and Complement, or Pseudorandom. The default is Complement. For more detail about these options, see “Data Pattern Test” on page 408. This option is only available if you have selected the data pattern test.
Pattern value	Specifies the pattern to use when performing a data pattern test. The default value is 0x55555555, which is a sequence of alternating binary zeros and ones. This option is only available if you have selected the data pattern test.
Memory read (nondestructive)	Selects the memory read test, which checks that each memory address obtains the same value when the address is read twice. For more details about this test, see “Memory Read Test” on page 411. This test cannot be run at the same time as any other memory tests.

CRC computation (nondestructive)	Selects the CRC computation test, which computes a CRC checksum on a range of memory. For more details about this test, see “CRC Compute” on page 411. This test cannot be run at the same time as any other memory tests.
CRC compare (nondestructive)	Selects the CRC compare test, which repeatedly computes CRC checksums for a range of memory, in order to verify that memory can be read reliably. For more details about this test, see “CRC Compare” on page 412. This test cannot be run at the same time as any other memory tests.
Find start/end ranges (nondestructive)	Selects the find start/end ranges test, which locates possibly unused memory at the start and end of the range being tested. For more details about this test, see “Find Start/End Ranges” on page 412. This test cannot be run at the same time as any other memory tests.

Setting Test Options

The third section of the **Perform Memory Test** window allows you to choose options that will apply to the tests you have selected. (Options that do not apply to the tests you have selected are dimmed.)

Test options

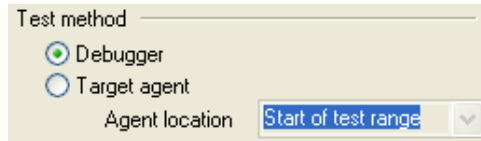
- ☐ Maximize address bus transitions
- ☐ Reset pattern value between iterations
- ☒ Repeat test (1-10000)
- ☐ Run test continuously
- ☒ Max. errors (1-1000)
- ☐ Write-only test (no errors)

The available options and their effects are listed in the following table. Not all options are available with all tests.

Maximize address bus transitions	Uses a sequence of addresses in the data pattern test or memory read test that maximizes the address line transitions between accesses. If this option is not selected, the default behavior accesses memory sequentially from low addresses to high addresses. When this option is selected, the sequence of memory accesses would begin <code>start</code>, <code>end</code>, <code>start+1</code>, <code>end-1</code>, and so on; assuming that <code>start</code> and <code>end</code> define a power of two sized and aligned region and the access size is one byte. If <code>start</code> and <code>end</code> do not define a power of two sized and aligned range, the range is split into sequential power of two sized and aligned ranges for the purposes of this test.
Reset pattern value between iterations	Executes the selected tests on every iteration rather than attempting to use different starting pattern values on successive test iterations. This option is only valid if the Repeat test value is greater than one or if the Run test continuously option is used. For pattern tests, this option has no effect if the Pattern option is Static.
Repeat test	Repeats memory test(s) the indicated number of times. If unspecified, the test(s) will be performed only once.
Run test continuously	Repeats memory test(s) continuously.
Max. errors	Aborts memory test(s) after detecting the specified number of errors. This value must be between 1 and 1000. If unspecified, MULTI will abort the test after 10 errors.
Write-only test (no errors)	Skips the reading phase of the address bus walking, data bus walking, and/or data pattern tests. Since no reading is performed, no errors can be reported.

Specifying Test Methods

This last section of the **Perform Memory Test** window allows you to choose whether tests will be performed by the Debugger reading and writing memory directly, or by MULTI downloading a small target agent program that will perform the memory accesses. (See “Memory Testing with a Target Agent” on page 403 for more details about using a target agent.)



The options available in this section and their effects are described in the table below.

Debugger	Specifies that the Debugger should perform the test or tests directly.
Target agent	Specifies that a target agent should be used to perform the test or tests. For more information, see “Memory Testing with a Target Agent” on page 403.
Agent location	Specifies where the target agent is downloaded. Select Start of test range or End of test range , or specify another location by entering it in the field, replacing the <specify location> choice. The default is to download the target agent at the start of the address range being tested. For more details, see “Memory Testing with a Target Agent” on page 403.

Running Memory Tests

After you have specified the memory range to be tested, the access size, the test(s) to perform, your test options, and your desired test method in the **Perform Memory Test** window, as described in the previous sections, click the **Test memory** button to run the selected test(s). The test(s) run on the target that was selected in the target list when you opened the **Memory Test Wizard**.

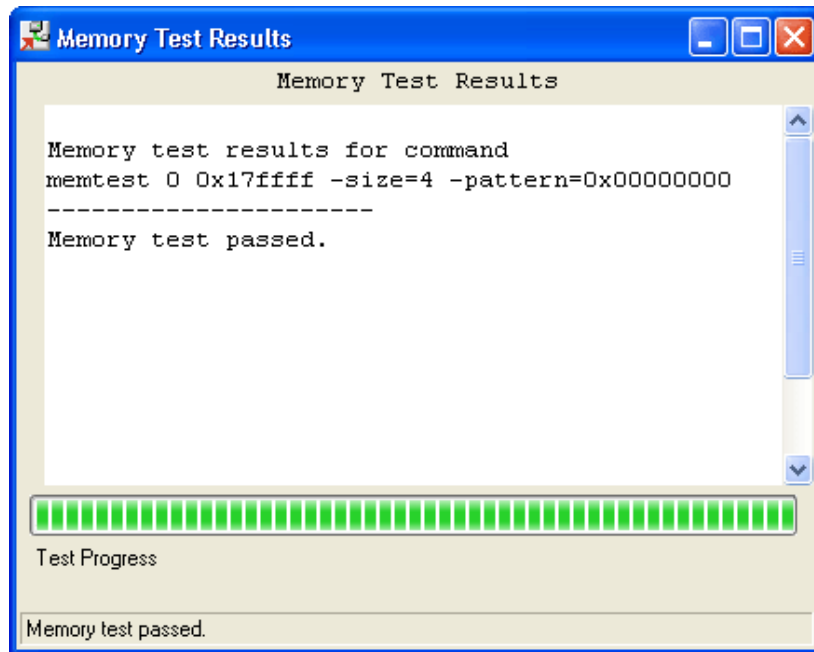
A **Memory Test Results** window is displayed and shows the test progress and results. For more information, see “Viewing Memory Test Results” on page 402. This window remains open after the test completes or is aborted.



Note To abort memory testing, press the Esc key.

Viewing Memory Test Results

After you launch one or more memory tests using the **Memory Test Wizard** or the **Perform Memory Test** window, a **Memory Test Results** window will appear.



The **Memory Test Results** window will contain the following information:

- A **memtest** command that is equivalent to the various tests and options selected in the **Memory Test Wizard** or **Perform Memory Test** window. (This is provided to make it easier to write MULTI Debugger scripts that can automatically perform memory tests.) For information about the **memtest** command, see “General Memory Commands” in Chapter 11, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.
- The printed output of the command, including any memory test errors. (This appears below the line of dashes.)
- A test progress indicator. (This appears below the window.)
- A status bar, which displays messages about the success or failure of the test(s).

Continuous Memory Testing

Continuous memory testing can be helpful when you are debugging hardware problems with a logic analyzer or other diagnostic equipment. To perform continuous testing, select the **Run test continuously** option in the **Perform Memory Test** window, or pass the **-continuous** option to the **memtest** command. To terminate the test after a specific number of errors, enter the number of errors in the **Max. errors** field, or pass the **-maxerr=number_of_errors** option to the **memtest** command. For information about the **memtest** command, see “General Memory Commands” in Chapter 11, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.

To terminate a continuous test manually, press Esc.

Memory Testing with a Target Agent

You can perform memory tests using either the Debugger or a small target agent program. Running memory tests from the Debugger is slower than running the tests using a target agent, but a target agent requires that the memory be initialized correctly. To specify which method you want MULTI to use, select either **Debugger** or **Target agent** in the bottom section of the **Perform Memory Test** window (see “Specifying Test Methods” on page 400).

When using a target agent, you must specify where MULTI should load the target agent code. To do this, use the **Agent location** field of the **Perform Memory Test** window to select one of the following locations for target agent placement:

- Start of test range (This is the default location)
- End of test range
- Specified location outside the test range



Note Keep the following restrictions and conditions in mind when specifying the location of your target agent:

- If you select the start or end of the test range, the test range must be at least twice as large as the target agent. This allows the complete range to be tested, by first placing the target agent at the start of the range, and then placing the target agent at the end of the range. (The target agent requires an

estimated 10–20 KB of memory for its code and data. However this varies by target CPU architecture and is subject to change.)

- The target agent should not be located at the start or end of the test range for nondestructive tests, or else the target agent will overwrite that memory.
- If you specify a target agent location other than the start or end of the test range, it must not overlap the test range.

If you are running memory tests using the **memtest** command, use the **-tgtagent** option to specify that memory testing be performed with a target agent. To specify the location of the target agent code, use one of the following options:

- **-tgtagentstart** — Place target agent at the start of the test range.
- **-tgtagentend** — Place target agent at the end of the test range.
- **-tgtagentloc=expr** — Place target agent at the location specified by the expression *expr*.

For more information about the **memtest** command, see “Running Memory Tests from the Command Line” on page 414.

Types of Memory Tests

The sections below describe the different types of memory testing in detail. For information about launching these tests, see “Quick Memory Testing: Using the Memory Test Wizard” on page 390, “Advanced Memory Testing: Using the Perform Memory Test Window” on page 394, or “Running Memory Tests from the Command Line” on page 414.

Address Bus Walking Test

The address bus walking test is a destructive memory test that provides an efficient way to verify that address lines are not stuck at one or zero and are not tied together. This test is usually fast enough to be run from the Debugger, so it generally is not necessary to use a target agent (see “Specifying Test Methods” on page 400 for information about these two test methods).

In this test, the range of memory is divided into sequential regions that are sized and aligned to a power of two. For example, if memory from 0xa00 to 0xffffffff is to be tested, the actual ranges that will be tested are as follows:

0xa00 to 0xbff (512 B)	0b1010 0000 0000 0b1011 1111 1111
0xc00 to 0xfff (1KB)	0b1100 0000 0000 0b1111 1111 1111
0x1000 to 0x1fff (2KB)	0b0001 0000 0000 0000 0b0001 1111 1111 1111

and so on, up to:

0x800000 to 0xffffffff (8MB)	0b1000 0000 0000 0000 0000 0000 0b1111 1111 1111 1111 1111 1111
---------------------------------	--

This test can be performed as a *walking one test* or a *walking zero test*, or you can set it to run twice, testing once for ones and once for zeros. Depending on the walking value selected (one or zero), a pattern value is written to an address in the range with a single address bit set to 1 (walking one test) or to 0 (walking zero test). All other address bits that reference that region are set to the opposite value. The low bits are cleared, as appropriate for the access size used. The pattern's complement is written to locations in the test range with a different address bit set (walking one test) or cleared (walking zero test). The pattern's complement is also written to the location at the start of the range (walking one test) or the end of the range (walking zero test). After the complement values are written, the original pattern value that was written is verified to make sure it has not changed. This is repeated for every address bit that references the range of memory to be tested.

The default pattern is 0x55555555, 0x5555, or 0x55, depending on the access size.

For example, using the range 0x0000 to 0xffff, a walking one test with an access size of 4 produces a sequence that begins as follows.

```
Write 0x55555555 to address 0x0004
Write 0xaaaaaaaa to address 0x0008
Write 0xaaaaaaaa to address 0x0010
...
Write 0xaaaaaaaa to address 0x8000
Write 0xaaaaaaaa to address 0x0000
Read from address 0x0004 and verify that the value is 0x55555555.
```

```
Write 0x55555555 to address 0x0008
Write 0xaaaaaaaa to address 0x0004
Write 0xaaaaaaaa to address 0x0010
...
Read from address 0x0008 and verify that the value is 0x55555555.
...
Write 0x55555555 to address 0x0000
Write 0xaaaaaaaa to address 0x0004
Write 0xaaaaaaaa to address 0x0008
...
Read from address 0x0000 and verify that the value is 0x55555555.
```

To run this test from the **Perform Memory Test** window, select the **Address walking** radio button, then select a **Walking value (One, Zero, or Both)**. See “Advanced Memory Testing: Using the Perform Memory Test Window” on page 394.

This test is also run twice, once with walking ones and once with walking zeros, if you select the **Quick** or **Exhaustive** buttons on the **Memory Test Wizard**. See “Quick Memory Testing: Using the Memory Test Wizard” on page 390.

To run this test using the **memtest** command, use the **-test=a0** option (for a walking zero test) and/or the **-test=a1** options (for a walking one test). See “Running Memory Tests from the Command Line” on page 414.

Data Bus Walking Test

The data bus walking test is a destructive memory test that provides a way to verify that data bus lines are not stuck at one or zero and are not tied together. This test is usually fast enough to be run from the Debugger, so it generally is not necessary to use a target agent (see “Specifying Test Methods” on page 400 for more information about these different approaches).

This test can be performed as a *walking one test* or a *walking zero test*, or you can set it to run twice, testing once for ones and once for zeros. Depending on the walking value selected (one or zero), successive values containing either a single one bit (walking one test) or a single zero bit (walking zero test) are written to successive memory locations and then read back. Only the first 8, 32, or 128 bytes of memory in the range are modified by this test, depending on the access size specified.

For example, using the range 0x0000 to 0xffff, a data walking one test with an access size of 4 produces a sequence that begins as follows:

```
Write 0x00000001 to address 0x0000
Write 0x00000002 to address 0x0004
Write 0x00000004 to address 0x0008
...
Write 0x80000000 to address 0x007c
Read from address 0x0000 and verify that the data value
    is 0x00000001
Read from all other addresses written and verify that the
    value is as expected.
```

You can specify a repeat count, which will rotate the initial data pattern at the start of each test. Using the same options utilized in the previous example, a repeat count will produce a sequence that begins with the following:

```
Iteration 1:
Write 0x00000001 to address 0x0000
...
Iteration 2:
Write 0x00000002 to address 0x0000
...
Iteration 3:
Write 0x00000004 to address 0x0000
...
```

If the memory range is not large enough to write all distinct values, the test will write some of the walking data values, read them back to verify that the value is what is expected, then start from the beginning of the memory range and write more of the walking data values.

To run this test from the **Perform Memory Test** window, select the **Data walking** radio button, then select a **Walking value (One, Zero, or Both)**. See “Advanced Memory Testing: Using the Perform Memory Test Window” on page 394.

This test is also run twice, once with walking ones and once with walking zeros, if you select the **Quick** or **Exhaustive** buttons on the **Memory Test Wizard**. See “Quick Memory Testing: Using the Memory Test Wizard” on page 390.

To run this test using the **memtest** command, use the **-test=d0** option (for a walking zero test) and/or the **-test=d1** options (for a walking one test). See “Running Memory Tests from the Command Line” on page 414.

Data Pattern Test

The data pattern test is a destructive memory test that provides a way to comprehensively test that all memory addresses in the range can successfully store values. This test can help diagnose ground bounce and voltage problems in the memory range.

The data pattern test writes a specified pattern to successive memory locations. The same pattern value can be written to all locations, or you can specify that certain modifications be made between iterations. The options for how the pattern is handled are listed in the table below.

Pattern option	Effect
Static	Writes the same value to all locations.
Complement	The specified pattern is complemented after each write. You can use this option to generate a large number of address and data bus transitions on successive memory accesses. For example, if the range 0x0000 to 0xffff is tested with the pattern 0x01234567, an access size of 4, and the pattern is complemented, the test sequence for the pattern test would be: <pre>Write 0x01234567 to address 0x0000 Write 0xfedcba98 to address 0x0004 Write 0x01234567 to address 0x0008 ... Read from all addresses written and verify that the value is as written.</pre>
Rotate	The specified pattern is rotated each time it is written. The previous pattern value is shifted one bit position to the left and the most significant bit of the previous pattern value is copied to the least significant bit position of the next pattern value.

Pattern option	Effect
Rotate and Complement	The specified pattern is rotated after its value and its complement are written. For example, if the range 0x0000 to 0xffff is tested with the pattern 0x01234567, an access size of 4, and the pattern is rotated and complemented, the test sequence for the pattern test would be: <pre> Write 0x01234567 to address 0x0000 Write 0xfedcba98 to address 0x0004 Write 0x02468ace to address 0x0008 Write 0xfdb97531 to address 0x000c ... Read from all addresses written and verify that the value is the same as was written.</pre>
Pseudorandom	The specified value is fed through a linear feedback shift register (LFSR) that generates a sequence of $2^n - 1$ values. If this method is selected, the next value in a sequence is generated as follows (bit positions are indicated with bit 0 as the least significant bit): 1. Compute a new bit 0 value by performing the exclusive-or of four bits in the current value. Bit positions in this description are numbered starting with the least significant bit. 2. Shift the current value left by one. 3. Insert the computed bit 0 value in the new value. The four bit positions vary according to access size, as follows: • If the access size is 8, the bit positions are 3, 4, 5, and 7. • If the access size is 16, the bit positions are 3, 12, 14, and 15. • If the access size is 32, the bit positions are 0, 1, 21, and 31. The data pattern consisting of all zeros is not permitted, since the LFSR will always generate a zero in that case.



Note You can use the **Maximize address bus transitions** option in the **Perform Memory Test** window or the **-maxtransitions** option with the **memtest** command (see “Running Memory Tests from the Command Line” on page 414) to cause MULTI to use a sequence of addresses for this test

that maximizes the address line transitions between accesses. If this option is not specified, the default behavior accesses memory sequentially from low addresses to high addresses. When this option is selected, the sequence of memory accesses would begin `start`, `end`, `start+1`, `end-1`, and so on; assuming that `start` and `end` define a power of two sized and aligned region and the access size is one byte. If `start` and `end` do not define a power of two sized and aligned range, the range is split into sequential power of two sized and aligned ranges for the purposes of this test.

To run this test from the **Perform Memory Test** window, select the **Data pattern** radio button, then select a **Pattern option** (**Static**, **Complement**, **Rotate**, **Rotate and Complement**, or **Pseudorandom**) and specify a **Pattern value**. See “Advanced Memory Testing: Using the Perform Memory Test Window” on page 394.

This test is also run, using a target agent, the pseudorandom pattern option, and the **Maximize address bus transitions** option, if you select the **Exhaustive** button on the **Memory Test Wizard**. See “Quick Memory Testing: Using the Memory Test Wizard” on page 390.

To run this test using the **memtest** command, use the **-test=p** option and specify the pattern with the **-pattern=value** option. To specify the pattern behavior, use the following options:

- **-complement** — To complement the data pattern value between memory writes.
- **-rotate** — To rotate the data pattern for the pattern test between writes.
- **-random** — To use a pseudorandom sequence of values for the pattern test.

You can pass both the **-complement** and **-rotate** options together. See “Running Memory Tests from the Command Line” on page 414.

Memory Read Test

The memory read test is a nondestructive memory test that verifies that each memory address retains the same value when the address is read twice.

This test reads memory from successive memory locations. After reading each address for the first time, the test reads back the value from the prior address and verifies that the values are identical.

This test does not destroy the memory in the range. For example, to test the range 0x0000 to 0xffff with an access size of 4, the test sequence begins:

```
Read from address 0x0000
Read from address 0x0004
Check: Read from address 0x0000 and verify that the value
       matches the earlier read.
Read from address 0x0008
Check: Read from address 0x0004 and verify that the value
       matches the earlier read.
...
```

To run this test from the **Perform Memory Test** window, select the **Memory read (nondestructive)** radio button. See “Advanced Memory Testing: Using the Perform Memory Test Window” on page 394.

This test is also run if you select the **Memory read (nondestructive)** buttons on the **Memory Test Wizard**. See “Quick Memory Testing: Using the Memory Test Wizard” on page 390.

To run this test using the **memtest** command, use the **-test=r** option. See “Running Memory Tests from the Command Line” on page 414.

CRC Compute

CRC compute provides a way to compute a CRC checksum on a range of memory. This nondestructive test reads bytes from successive memory locations and computes a CRC checksum from the result. The algorithm used is a standard 32-bit CRC and matches the algorithm used by default in the Green Hills Software linker’s **-checksum** option.

To run this test from the **Perform Memory Test** window, select the **CRC computation (nondestructive)** radio button. See “Advanced Memory Testing: Using the Perform Memory Test Window” on page 394.

To run this test using the **memtest** command, use the **-test=cr** option. See “Running Memory Tests from the Command Line” on page 414.

CRC Compare

CRC compare provides a way to repeatedly compute a CRC checksum for a range of memory, in order to verify that memory can be read reliably. This nondestructive test reads bytes from successive memory locations and computes a CRC checksum from the result. It then recomputes the CRC checksum and verifies that the recomputed checksum matches the original computed value. You can specify how many times the test should be repeated. If the checksum does not match, an error is printed and the most recent checksum value is used from that point on for any further verify iterations.

The algorithm used is a standard 32-bit CRC and matches the algorithm used by default in the Green Hills Software linker’s **-checksum** option.

To run this test from the **Perform Memory Test** window, select the **CRC compare (nondestructive)** radio button. To run the test repeatedly, also select the **Repeat test** radio button and specify the number of times the test should run. See “Advanced Memory Testing: Using the Perform Memory Test Window” on page 394.

To run this test using the **memtest** command, use the **-test=cc** option. See “Running Memory Tests from the Command Line” on page 414. To cause the CRC compare to run more than once, use the **-repeat=number_of_tests** to specify the number of times the operation will be performed. See “Running Memory Tests from the Command Line” on page 414.

Find Start/End Ranges

The find start/end ranges test is a nondestructive test used to discover possibly unused memory at the start and end of the specified range. Sequences of identical values of the specified access size at the start or end of the specified range, if present, are reported as possibly unused memory.



Note The term *range* is used in different ways in this test. The *specified range* represents the entire area of memory to be examined and is an input to this test. The *start range* and *end range*, if present, are subranges of the specified range and are the outputs of this test.

This test consists of two parts. First, the test reads memory locations sequentially from the start of the specified range. If multiple consecutive memory locations contain the same value, those consecutive, identical memory locations are reported as a potentially unused memory range. When the test reads the first memory location that contains a value different from all the earlier values, the first part of the test stops. The second part of the test is similar to the first part of the test, except that memory locations are read sequentially from the end of the specified range toward the start of the specified range. The output indicates the extent of identical memory values at the start and end of the provided range. (If the entire specified range contains the same value, the test will only report a single range equal to the specified range.)

For example, if the range 0x0000 to 0xffff is tested with an access size of 4, the test output might be something like:

```
Start range: 0x00000000
           to: 0x000000ff
           value: 0x00000000

End range: 0x0000ba4c
           to: 0x0000ffff
           value: 0xffffffff
```

This output indicates that the first 0x100 bytes of the range contain the 4-byte value 0x00000000. This output also implies that the next memory location contains a value other than 0x00000000. The final 0x45b4 bytes of the specified range (0xba4c-0xffff) contain the 4-byte value 0xffffffff. This output also implies that the location 0xba48 contains a value other than 0xffffffff.

To run this test from the **Perform Memory Test** window, select the **Find start/end ranges (nondestructive)** radio button. See “Advanced Memory Testing: Using the Perform Memory Test Window” on page 394.

To run this test using the **memtest** command, use the **-test=fr** option. See “Running Memory Tests from the Command Line” on page 414.

Efficient Testing Methods

MULTI provides a wide variety of memory testing options ranging from simple to complex. You may want to devise a testing scheme to reap the most benefit from the various testing options. For instance, you might first use Debugger-based tests to verify that memory is initialized correctly and that there are no problems with data or address lines, and then move on to running tests using a target agent, in order to obtain broader testing coverage. A sample testing sequence using this model might be:

1. Perform one or more address and/or data walking tests from the Debugger.
2. Perform one or more pattern tests over a small area of memory from the Debugger.
3. Perform one or more of the other tests using the target agent across the entire range of memory.

Running Memory Tests from the Command Line

Memory testing can be performed using the **memtest** command. However, due to the vast array of test types and options, the syntax for this command can be quite complex. For this reason, we recommend that you use the **Memory Test Wizard** or the **Perform Memory Test** window to configure and run memory tests.



Note If you want to write scripts that contain memory testing commands, you can still use the **Memory Test Wizard** or the **Perform Memory Test** window to help you determine the exact command syntax for the specific test and testing options you want to use. To do this, first use one of these graphical tools to configure the test you want to run, and then run the test. When the test completes, the **Memory Test Results** window will display the exact command syntax that corresponds to the testing options you specified in the GUI. You can use the command syntax given there in the Debugger command pane or in customized scripts.

The basic syntax for the **memtest** command follows. For further details about the test types and the options mentioned, see the fuller descriptions in “Advanced Memory Testing: Using the Perform Memory Test Window” on page 394 and “Types of Memory Tests” on page 404.

memtest *start_addr end_addr* -size=*size* -test=*test_choice*
[-pattern=*value*] [-complement | -rotate | -random | -complement
-rotate] [-maxtransitions] [-resetpattern] [-repeat=*number_of_tests*
| -continuous] [-maxerr=*number_of_errors* | -writeonly] [-tgtagent]
[-tgtagentstart | -tgtagentend | -tgtagentloc=*expr*]

where:

- *start_addr* is an expression that defines the lowest address to test. The expression must evaluate to a 32-bit address.
- *end_addr* is an expression that defines the highest address to test. The expression must evaluate to a 32-bit address.
- *size* indicates the access size (in bytes) to use when performing the test and can be 1, 2, or 4.
- **-test=test_choice** defines the type of test to run. Acceptable values for *test_choice* are:
 - **a1** — Address walking one test (destructive)
 - **a0** — Address walking zero test (destructive)
 - **d1** — Data walking one test (destructive)
 - **d0** — Data walking zero test (destructive)
 - **p** — Data pattern test (destructive)
 - **r** — Memory read test (nondestructive)
 - **cr** — CRC computation (nondestructive)
 - **cc** — CRC compare test (nondestructive)
 - **fr** — Find start/end ranges test (nondestructive)

More than one of the destructive memory test options (**a1**, **a0**, **d1**, **d0**, and **p**) can be specified by using multiple **-test=test_choice** arguments. The nondestructive tests (specified with the values **cc**, **cr**, **r**, or **fr**) must be performed individually.

For full descriptions of these test types, see “Types of Memory Tests” on page 404.

- **-pattern=value** — Specifies the data value to use for address bus walking and/or data pattern tests.

- **-complement** — Causes the data pattern value for pattern tests to be complemented between memory writes. (This option can be passed with the **-rotate** option. If both **-complement** and **-rotate** are specified, the pattern will be rotated every other write and complemented every write, resulting in a write sequence similar to 0x01, 0xfe, 0x02, 0xfd, . . .)
- **-rotate** — Causes the data pattern for pattern tests to be rotated between writes. (This option can be passed with the **-complement** option as well, causing the pattern value to be both complemented and rotated between iterations. See the description for **-complement** above.)
- **-random** — Causes a pseudorandom sequence of values to be used for the pattern test. (This option cannot be used with either the **-complement** or the **-rotate** options.)
- **-maxtransitions** — Causes MULTI to use a sequence of addresses in the data pattern or memory read tests that maximizes the address line transitions between accesses. (The default behavior is to access memory sequentially from low addresses to high addresses.)
- **-repeat=number_of_tests** — Specifies the number of times to repeat a test or tests. (This option cannot be passed with the **-continuous** option, and does not apply to the CRC compute or find start/end ranges tests.)

If a **-repeat** value is specified with multiple memory tests, all selected tests are run during each iteration. For example, the options **-test=a0 -test=p -repeat=2** would set the test sequence as:

```
Address walking zero
Pattern
Address walking zero
Pattern
```

- **-resetpattern** — Causes MULTI to use the same starting pattern value for each test iteration rather than using a complemented, rotated, or subsequent pseudorandom value.
- **-continuous** — Specifies that the test(s) will run continuously. (This option cannot be passed with the **-repeat=number_of_tests** option, and does not apply to the CRC compute or find start/end ranges tests.)
- **-maxerr=number_of_errors** — Causes MULTI to abort the test(s) after detecting the specified number of errors. This option cannot be used with the **-writeonly** option.

- **-writeonly** — Causes MULTI to skip the reading phase of the address bus walking, data bus walking, and data pattern tests. This option cannot be used with the **-maxerr=number_of_errors** option.
- **-tgtagent** — Causes MULTI to download and use a target-resident agent to perform the memory tests (rather than using the Debugger).
- **-tgtagentstart** — Specifies that the target agent be placed at the start of the memory range to be tested. This option is only valid when used with the **-tgtagent** option.
- **-tgtagentend** — Specifies that the target agent be placed at the end of the memory range to be tested. This option is only valid when used with the **-tgtagent** option.
- **-tgtagentloc=expr** — Specifies that the target agent be placed at the location specified by the expression *expr*. This memory location should not overlap the test range, and must be valid for the test to be performed successfully.

Detecting Coherency Errors

If the contents of memory differ from what you loaded onto your target, a coherency error may exist. In this context, *coherency* refers to the byte-value agreement between memory and the file that you originally loaded. For example, most programs contain a `.text` section that is not modified. As a result, the byte values of the `.text` section that appears in the executable program file should match the byte values of the `.text` section that is loaded into memory while the process is running.

Coherency errors may occur as the result of self-modifying code, buffer or stack overruns, bad memory hardware, or simply the wrong version of code being debugged. In the worst case scenario, code in memory is only slightly different from code in the executable program file, making it difficult to detect the discrepancy. MULTI offers two complementary ways of detecting this problem: manual coherency checking via the **verify** command and automatic coherency checking via a debugging setting. The next two sections describe each of these coherency checking methods.



Note MULTI supports coherency checking with most debug servers. However, if the **Debug** → **Debug Settings** → **Auto Check Coherency** menu item is dimmed, most automatic coherency checking is not available in your current environment.

Checking Coherency Manually

To manually check the coherency of an address range, enter the following command in the command pane:

verify *address_expression* [*num_addresses*]

where:

- *address_expression* specifies the address expression at which to begin coherency checking.
- *num_addresses* specifies the number of addresses to verify past *address_expression*. If you omit *num_addresses*, this command verifies until the end of the function that encloses *address_expression*.

MULTI compares the bytes in the specified range of memory and the bytes in the content of the executable program file. It then prints a list of differing addresses. Additionally, it highlights lines containing addresses that differ. (Highlighting is done in the source pane.)

To manually check the coherency of all downloaded sections that cannot be written to, enter the **verify -all** command in the command pane. The `.text` section is one example of a section that you cannot write to. Because certain sections of memory, such as `.bss`, `.data`, and `.heap`, may be written to during program execution, you can expect them to differ from the executable program file. When you specify the **-all** option, the **verify** command does not check these sections. However, you can verify them manually by entering **verify -section** *section_name* in the command pane.

For comprehensive information about the **verify** command, see Chapter 11, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.

Checking Coherency Automatically

When automatic coherency checking is enabled, MULTI checks the coherency of the specified number of addresses at every stop. If it finds discrepancies, it highlights the differing lines in the source pane.

To enable automatic coherency checking, do one of the following:

- Select **Debug → Debug Settings → Auto Check Coherency**.

- Set the `_AUTO_CHECK_COHERENCY` system variable. Set this variable to nonzero to enable automatic checking; set it to zero to disable automatic checking. See also the `_AUTO_CHECK_COHERENCY` variable in “System Variables” on page 291.



Note The Debugger always compares user-initiated memory writes with the expected contents of memory and highlights any coherency errors this causes, regardless of whether or not automatic coherency checking is enabled. For example, if the user writes `0xdeadbeef` to the first instruction of function `foo()`, the Debugger highlights the source line associated with that instruction because the memory on the target no longer matches what is expected to be there.

MULTI attempts to check an equal number of addresses before and after the current program counter; however, it does not cross procedure boundaries. For example, if MULTI is stopped at the first instruction of a procedure, automatic checking inspects addresses only after the program counter. It continues for the specified number of addresses.

By default, MULTI checks either 16 addresses or the number of addresses lasting the length of 4 instructions (whichever is fewer). To change the number of addresses checked, do one of the following:

- Select **Debug** → **Debug Settings** → **Number Of Addresses To Check**. In the dialog box that appears, enter the number of addresses you want checked on each stop.
- Set the `_AUTO_CHECK_NUM_ADDRS` system variable to an appropriate value. See also the `_AUTO_CHECK_NUM_ADDRS` variable in “System Variables” on page 291.



Note Because extra memory is read on every stop, use care when setting the number of addresses to be read. Setting a large number of addresses may slow normal run control.

Run-Mode Debugging

Chapter 18

This Chapter Contains:

- Establishing Run-Mode Connections
- Loading a Run-Mode Program
- The Task Manager
- Working with AddressSpaces and Tasks
- Working with Task Groups in the Task Manager
- Working with Run-Mode Breakpoints
- Task Manager Options
- Task Group Configuration File
- OS-Awareness in Run Mode
- Profiling All Tasks in Your System

When you debug a multitasking application, MULTI interacts with the target's operating system in run mode, in freeze mode, or in both run and freeze mode simultaneously. In run mode, the operating system kernel continues to run as you halt and examine individual tasks. In freeze mode, the entire target system stops when you examine tasks. For information about freeze-mode debugging, see Chapter 19, "Freeze-Mode Debugging and OS-Awareness". For information about debugging in run mode and freeze mode, see "Freeze-Mode and Run-Mode Connections to the Same Target" on page 70.

MULTI supports run-mode debugging for special target environments comprised of specific processor/RTOS/debug server combinations. The debug server must support multitasking applications.



Note In this chapter, *task* is used as a general term to describe the real-time operating system's unit of execution. Specific terminology varies according to the target's operating system (for example, a task may also be called a *process* or a *thread*).

Establishing Run-Mode Connections

Before utilizing the run-mode debugging features described in this chapter, you must establish a run-mode connection to your target via a debug server that supports multitasking debugging, such as **rtserv** or **rtserv2**. For general information about connecting, see Chapter 4, "Connecting to Your Target". For specific information about connecting with **rtserv2**, see "Connecting with INDRT2 (rtserv2)" in the *INTEGRITY Development Guide*. For information about connecting with other debug servers, see the *MULTI: Configuring Connections* book for your target processor family.

After you have connected to your target, the connection appears in the target list. It's usually denoted by an entry labeled **Run mode target**. For information about the way in which run-mode AddressSpaces and tasks appear in the target list, see "Debugging Target List Items" on page 34.



Note If you want to simultaneously debug multiple targets in run mode, use one instance of MULTI to establish the corresponding run-mode connections. Distinct targets and applications are accessible for debugging via the Debugger's target list (see "The Target List" on page 33). If you launch multiple instances of MULTI (for example, by double-clicking the MULTI shortcut twice or by launching MULTI from the command line twice), the results of operations between MULTI IDE tools is undefined.

Loading a Run-Mode Program

Some run-mode targets, including INTEGRITY, allow you to dynamically download programs onto the target after it has booted.

To dynamically download a program, do one of the following:

- In the Debugger, select **Target** → **Load Module**.
- In the Task Manager, select **Load** → **Load Module**.
- In the Debugger command pane, enter the **load** command. For information about this command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.

The exact requirements for loading a module depend on your target operating system. If you are running INTEGRITY, your target must be configured with a dynamic loader. For more information, see “Dynamic Downloading” in the *INTEGRITY Development Guide*. For general information about debugging dynamically downloaded INTEGRITY applications, see “Run-Mode Debugging” in the *INTEGRITY Development Guide* and “Troubleshooting” in the *INTEGRITY Development Guide*.



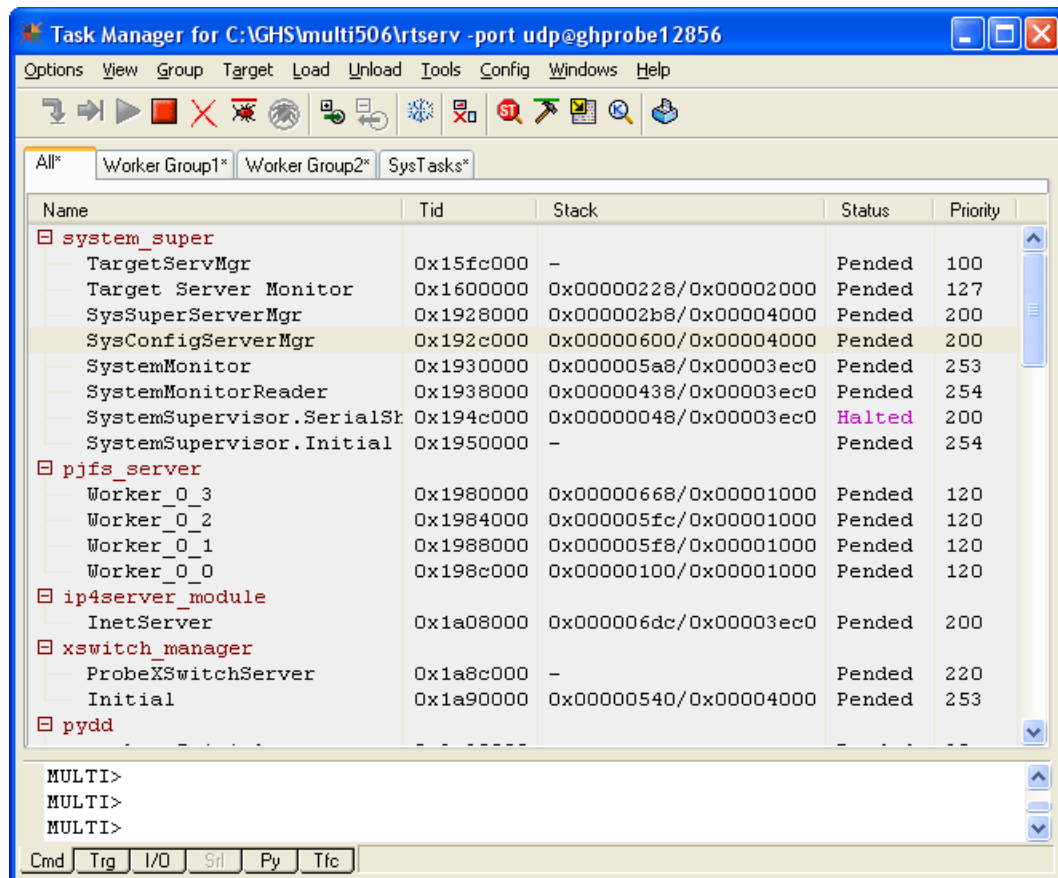
Note Depending on your configuration, MULTI may attempt to restore breakpoints saved from previous debugging sessions when you download a program. (See the description of the **rememberBreakpoints** option in “Session Configuration Options” in Chapter 9, “Configuration Options” in the *MULTI: Editing Files and Configuring the IDE* book.) INTEGRITY targets also allow programs to specify one or more tasks that may automatically start as soon as the program has finished downloading. Note that INTEGRITY versions 5 and earlier do not guarantee that MULTI will be able to install breakpoints on the target before tasks are allowed to begin execution. If you are using INTEGRITY version 5 or earlier and you want breakpoints to be restored, perform one of the following operations to prevent tasks from starting automatically:

- In the applicable Task section(s) of your Integrate configuration file (**.int**), set **StartIt** to **false**. Then rebuild your application.
- Before downloading your program in MULTI, enable **Debug** → **Target Settings** → **Stop on Task Creation**.

The Task Manager

During a run-mode debugging session, you can use the Task Manager to work with tasks created by your application. After connecting to a debug server that supports multitasking debugging (see “Establishing Run-Mode Connections” on page 422), do one of the following to open the Task Manager:

- In the **Connection Organizer**, select **Target** → **Show Task Manager**.
- In the Debugger, select **View** → **Task Manager**.
- In the Debugger command pane, enter the **taskwindow** command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.
- In the Project Manager, select **Connect** → **Task Manager**.



Each tab of the Task Manager corresponds to a *task group* (see page 428).

Working with AddressSpaces and Tasks

Run-Mode AddressSpaces

Run-mode AddressSpaces are located in the target list under a CPU node of a run-mode connection to the target, or they are located under an application (kernel image or dynamic downloaded module). The application is in turn located under a CPU node. A run-mode connection to a target is usually denoted by **Run mode target**.

If you are using INTEGRITY version 10 or later and you select a run-mode AddressSpace in the target list, the AddressSpace's source code appears in the source pane. In general, the following operations are available.

- Source browsing
- Breakpoint manipulation (All breakpoints set here are any-task breakpoints. For more information, see “Any-Task Breakpoints” on page 430.)

If you are not using INTEGRITY version 10 or later and you select a run-mode AddressSpace in the target list, the source pane is empty and the preceding operations are unavailable.

Attaching to Tasks

Attaching to a task allows MULTI to obtain debugging access to that task. When you attach to a task, MULTI automatically displays the task in a Debugger window.

Some operating systems require that you attach to a task before halting, killing, or running it, while other operating systems allow you to perform these actions on unattached tasks.

To attach to a task, do one of the following:

- In the Task Manager, double-click the task.
- In the Debugger's target list, select the task. (For information about identifying run-mode tasks in the target list, see “Debugging Target List Items” on page 34.)



Note When you have attached to a task that is listed as **Halted** in the target list's **Status** column, you should avoid using programmatic means (for example, INTEGRITY's `RunTask()` kernel call) to cause that task to run. MULTI is not guaranteed to detect such target-initiated status changes. To work around this problem, issue the **detach** command to detach from the task, preferably before the target causes it to run. For information about the **detach** command, see Chapter 3, "General Debugger Command Reference" in the *MULTI: Debugging Command Reference* book.

Halting Tasks On Attach

Some operating systems require that you attach to a task before halting it. To automatically halt a task as soon as you attach to it, do one of the following:

- In the Task Manager, select **Options** → **Halt on Attach**.
- In the Debugger, select **Debug** → **Target Settings** → **Halt on Attach**.

When the attached task opens in the Debugger, it is already halted.

Debugging Run-Mode Tasks

When you attach to a task, MULTI automatically displays the task in the Debugger window. To automatically display tasks in the Debugger as soon as the application creates them, enable **Stop On Task Creation** and **Attach on Task Creation** from the Task Manager's **Options** menu or from the Debugger's **Debug** → **Target Settings** menu.

If you open a new Debugger window to debug each new task (see "Opening Multiple Debugger Windows" on page 32), MULTI color-codes each task according to its corresponding Debugger window. Any window that relates to debugging a particular task is shaded in the same color as the task is shaded in the Task Manager. For example, suppose you attach to a task that is shaded blue in the Task Manager. The Debugger window that shows the task's code is also shaded blue. If you use a Browse window to look at the procedures of the task, the Browse window is also shaded blue.

Once a task is open in a Debugger window, you can halt or run the task from the Debugger window itself or from the Task Manager. If a breakpoint is hit in a task other than the one currently selected in the target list, the Debugger switches to that task by default (provided that no other Debugger window is open on it and that the currently selected task is not halted). For information

about the configuration option that allows you to control this behavior, see **targetWindowSwitchViewOnBpHit** in “Other Debugger Configuration Options” in Chapter 9, “Configuration Options” in the *MULTI: Editing Files and Configuring the IDE* book.

Freezing the Task Manager’s Task List Display

Because run-mode debugging allows some tasks to continue running while you debug other tasks, the data in the Task Manager changes constantly. If you want to take a snapshot of all the tasks at a certain moment, choose **View → Freeze Task List**. This option freezes the task list display (see “Task List Pane” on page 440), not the underlying operating system. When the display is frozen, a message appears in the status bar at the bottom of the Task Manager. You can modify task groups while the window is frozen.



Tip To change the rate at which MULTI refreshes the tasks in an unfrozen Task Manager:

1. In the Debugger or Project Manager, select **Config → Options**.
2. In the **Options** window that appears, click the **Debugger** tab.
3. Click the **More Debugger Options** button.
4. Edit the **Interval to refresh Task Manager** field. For more information, see “The More Debugger Options Dialog” in Chapter 9, “Configuration Options” in the *MULTI: Editing Files and Configuring the IDE* book.

Working with Task Groups in the Task Manager

Task groups allow you to organize tasks, making it easier to work with multiple tasks simultaneously. In addition, task groups provide the approach for group breakpoints. (For information about group breakpoints, see “Group Breakpoints” on page 431.)



Note The Task Manager must be open in order to guarantee that task groups work well.

A task group can contain any task in the system, including tasks running on different processors or in different chunks of logical memory (for example, tasks running in different AddressSpaces of INTEGRITY). As you create task groups, they are displayed as new tabs in the Task Manager. The name of the task group must consist of one or more characters and cannot contain square brackets ([and]). In addition, you cannot name a task group with one of the default task group names created by MULTI (see next section).

To quickly create a task group that contains the appropriate tasks, select the tasks in the Task Manager and click .

You can attach to, halt, kill, and run all the tasks in a task group with a single action. The **Group** menu contains several options that act on all of the tasks in the current task group.

MULTI supports task groups whether or not the underlying debug server and debug agent on the target support task groups. The ability of the underlying debug server and target debug agent to support task groups affects synchronous operation latency. For more information, see “Group Breakpoints” on page 431 and “Synchronous Operations” on page 432.

Default Task Groups

MULTI creates the following two default task groups:

- **All** — Contains all tasks that the operating system is using to run the application. There are some internal RTOS tasks, however, that the operating system may not include in the **All** group.
- **System** — Contains all tasks running on the target system, including the internal RTOS tasks that the operating system excludes from the **All** group.

For some operating system/debug server combinations, the set of tasks contained in the **System** group is identical to the set of tasks contained in the **All** group.

In general, task group names are case-sensitive, but MULTI treats the two default groups as being case-insensitive. As a result, you cannot create a group named **All** or **System**, no matter what the case combination. The **All** group has a tab in the Task Manager, but the **System** group does not.

You cannot add tasks into or delete tasks from the **All** or **System** default groups.



Tip When both the operating system and the debug server support task groups and distinguish between the **System** and **All** default groups, halting the group **System** freezes the target board and all application tasks. In this environment, you can execute operations such as reliably dumping the event log section, browsing kernel objects with the **OSA Explorer**, etc. To halt the **System** task group, select **Group** → **Halt System**.

MULTI reserves the following task group names. They are not shown as tabs in the Task Manager, but you should not use them for groups you create:

- **Current Task** – A label used to represent the special task group containing only the corresponding task in the **Software Breakpoint Editor** (see “Creating and Editing Software Breakpoints” on page 106).
- **Group N/A** – A label used to indicate that task group is not available in the **Software Breakpoint Editor** (see “Creating and Editing Software Breakpoints” on page 106).
- **__multi_tmp_op_grp_*** – A temporary group name created by MULTI to perform synchronous operations on the selected task if the underlying debug server supports task groups. Whenever MULTI sends the synchronous operation to the underlying debug server, the temporary group is automatically destroyed. For more information, see “Synchronous Operations” on page 432.
- **__multi_tmp_bp_grp_*** – A temporary group name created by MULTI to set a group breakpoint on one task. The group only contains the corresponding task. When the corresponding group breakpoint is deleted, the temporary group is automatically destroyed.
- **All AddressSpace names for INTEGRITY** — On INTEGRITY (version 5.0 and after), AddressSpace names can be used as groups when using group

breakpoints. However, AddressSpace names cannot be used as groups when using the **groupaction** command, and you cannot add tasks into or delete tasks from any AddressSpace group via the Task Manager. (For information about the **groupaction** command, see Chapter 19, “Task Group Command Reference” in the *MULTI: Debugging Command Reference* book.)

Configuring Task Group Settings

For advanced configuration options, see “Task Group Configuration File” on page 444.

Working with Run-Mode Breakpoints

MULTI supports three types of run-mode breakpoints: task-specific breakpoints, any-task breakpoints, and group breakpoints. Each of these is described in the following sections.

Task-Specific Breakpoints

A task-specific breakpoint is a breakpoint that only a single task can hit. To set a task-specific breakpoint, perform the following steps:

1. Attach to a task by double-clicking it in the Task Manager or selecting it in the Debugger’s target list.
2. In the Debugger window, click a breakdot ().

A task-specific breakpoint is indicated by a normal breakpoint icon ().

Any-Task Breakpoints

An any-task breakpoint is a breakpoint that every task in the AddressSpace (except system tasks, which cannot be debugged in run mode) can hit. An any-task breakpoint is indicated by a breakpoint icon with the letters AT inside of it ().

You can set an any-task breakpoint in a run-mode AddressSpace (if the target RTOS is supported) or in a task. To set the breakpoint from a run-mode

AddressSpace, select the AddressSpace in the target list and then click a breakdot (•) in the Debugger window.

To set the breakpoint from a task, select the task in the target list and hold the Shift key as you click a breakdot (•) in the Debugger window.

To set an any-task breakpoint from the command pane, use the **sb at** command. For example:

```
sb at function#5
```

sets an any-task breakpoint at line 5 of the function *function*. For more information about the **sb** command, see Chapter 4, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

When a task hits an any-task breakpoint, all other running tasks continue to execute.

Group Breakpoints

A group breakpoint sets a breakpoint on a group of tasks such that any task in the group can hit the breakpoint. When a task hits the breakpoint, the task, the whole group, or another group of tasks stops. A group breakpoint is indicated by a breakpoint icon with the letters GT inside it (•^{GT}).

You can set a group breakpoint via the **Breakpoints Window** (see “Viewing Breakpoint and Tracepoint Information” on page 105) or via breakpoint commands.

For more information about group breakpoints, see the **b** and **d** commands in Chapter 4, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

In addition to saving time, applying an action to an entire task group allows you to perform synchronous actions on multiple tasks. As long as the target operating system supports task groups, these actions are performed on the individual tasks almost simultaneously (see “Synchronous Operations” on page 432). If an operating system does not support task groups, MULTI sends out separate commands to each task in the task group. In this case, the latency time for the operations on different tasks is unpredictable, depending on various factors such as network traffic, the RTOS debug agent’s status, and the target’s speed.



Tip Use the **setsync** command to associate task groups with a specific action (available actions are running, halting, and stepping). When the action is performed on the current task, the action is also performed on the tasks in the associated task groups. Use the **showsyntax** command to display which task groups are currently associated with an action. For more information about the **setsync** and **showsyntax** commands, see Chapter 19, “Task Group Command Reference” in the *MULTI: Debugging Command Reference* book.



Note The Debugger may not be able to restore saved group breakpoints.

Synchronous Operations

Synchronous operations run and/or halt a group of tasks *almost* at the same time. The accuracy of synchronous operations varies for different debugging environments. If the debug server and the corresponding RTOS debug agent support task groups (as is the case with INTEGRITY), accurate synchronous operations and group breakpoints can be achieved; otherwise, group breakpoints are not supported. In this case MULTI sends out separate commands to each task, and the latency time for the operations on different tasks is unpredictable (depending on various factors such as network traffic, the RTOS debug agent’s status, the target’s speed, etc.).

On an RTOS that supports task groups, a group is *fine* if it meets the following conditions:

- The group is correctly created on the debug server.
- Subsequent manipulations to the group (such as adding tasks to the group) are correctly performed by the debug server.

Manipulating a group may change its status. For example, if you are using a debug server that does not support tasks from different CPUs in the same group, and you try to add tasks from different CPUs into an existing fine group, the group’s status becomes *not fine*.

MULTI keeps a list of fine groups for each debug server. A fine group can be used to create a group breakpoint or to perform synchronous operations in a group breakpoint. If you try to add tasks into a fine group, but the manipulation fails (rejected by the debug server) and no group breakpoint is set on the group,

MULTI deletes the group from the list of fine groups and asks the debug server to destroy the group in its scope.

When performing operations on multiple tasks selected in the Task Manager, MULTI attempts to take advantage of task group support in the debug server and in the corresponding RTOS debug agent. For example, when you halt selected tasks, MULTI attempts to create a temporary task group for the selected tasks. If such a fine group can be created on the debug server, MULTI sends one command to the debug server for the entire group, thus guaranteeing synchronization accuracy by the debug server and the RTOS debug agent. When performing operations on multiple tasks selected in the target list, however, MULTI does not attempt to construct a fine group. Instead, MULTI sends separate commands for each task selected, performing run-control operations one at a time.

Setting Breakpoints for Task Groups

Task groups are very powerful when setting breakpoints. (For information about task groups, see “Working with Task Groups in the Task Manager” on page 428.) By using task groups, you can:

- Set a group breakpoint such that any task in the group can hit the breakpoint.

For example, to set a group breakpoint at address 0x1423 on the **Tasks in App1** task group, enter the following in the Debugger’s command pane:

```
b /type_gt @"Tasks in App1" 0x1423
```

When any task that belongs to the **Tasks in App1** task group hits the breakpoint, that task is stopped. You cannot use MULTI commands with a group breakpoint, and the breakpoint count (@bp_count) argument must be 1 (default).

- Stop an entire task group when a group breakpoint is hit.

For example, suppose you want to halt all tasks in the **Callers** task group whenever a task in the **Tasks in App1** task group hits a breakpoint. You would enter:

```
b /type_gt @"Tasks in App1" /trigger @"Callers" 0x1423
```

When any task that belongs to the **Tasks in App1** task group hits the breakpoint, that task, along with all tasks in the **Callers** task group, is

stopped. You can use the `/trigger` task group to simultaneously stop multiple tasks when a breakpoint is hit.

If you want to create a group breakpoint only on the current task to stop all tasks in another group, it is not necessary for you to explicitly create a task group only containing the current task. MULTI does so automatically, and when the breakpoint is removed, the corresponding temporary task group is automatically destroyed.

For example, suppose you want to halt all tasks in the **Callers** task group whenever the current task hits a breakpoint. You would enter:

```
b /type_gt /trigger @"Callers" 0x1423
```

If, as in the following command, you do not specify `/trigger @task_group`:

```
b /type_gt 0x1423
```

the effect of the group breakpoint is equivalent to the effect of a normal breakpoint.

- Use an individual breakpoint to invoke an action on an entire task group. For example, to create a breakpoint on the current task that will halt all tasks in the **Pizza Tasks** group, enter:

```
b main#21 {groupaction -h @"Pizza Tasks"}
```

The **groupaction** command also supports running tasks (`-r`) and, for some operating systems, stepping tasks (`-s`). For more information about this command, see Chapter 19, “Task Group Command Reference” in the *MULTI: Debugging Command Reference* book.



Note When synchronous group operations are specified in a breakpoint command list (as in the preceding example), the amount of time that elapses between hitting the breakpoint and executing the synchronous group operations is not predictable.

As long as the target operating system supports task groups, these breakpoint actions are performed on the individual tasks almost simultaneously. If an operating system does not support task groups, MULTI sends out separate commands to each task in the task group. In this case, the latency time for the

operations on different tasks is unpredictable, depending on various factors such as network traffic, the RTOS debug agent's status, and the target's speed.

You can also use task groups to quickly set a normal breakpoint on each individual task in a task group. For example, to set a breakpoint with a count of 2 on every task in the **Pizza Tasks** task group, enter:

```
b @2 @"Pizza Tasks" 0x1423
```

Because individual breakpoints are applied to the tasks rather than a group, you can specify a breakpoint count of 2. Whenever one of the tasks in **Pizza Tasks** hits the breakpoint twice, it stops.

You can also remove individual breakpoints from all the tasks in a task group. For example, to remove breakpoints at address 0x1423 from any task belonging to the **Pizza Tasks** group, enter:

```
d @"Pizza Tasks" 0x1423
```

You can set a group breakpoint from the **Breakpoints** window (see “Viewing Breakpoint and Tracepoint Information” on page 105). For general information about breakpoints, see Chapter 4, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

Task Manager Options

This section describes the individual options available in the Task Manager.



Note If a debugging environment does not support a certain option, that option is not displayed in the Task Manager.

Menu Bar

The Task Manager contains the following menus:

- The **Options**, **View**, and **Group** menus are described below.
- The **Target**, **Load**, and **Unload** menus contain menu items that are duplicated in the Debugger's **Target** menu. See “The Target Menu” on page 519.

- The **Tools** menu contains menu items that are duplicated in the Debugger's **Tools** menu. See "The Tools Menu" on page 523.
- The **Config** menu is the same as the Debugger's **Config** menu. See "The Config Menu" on page 525.
- The **Windows** menu is the same as the Debugger's **Windows** menu. See "The Windows Menu" on page 530.
- The **Help** menu contains menu items that are duplicated in the Debugger's **Help** menu. See "The Help Menu" on page 530. In addition, the **Task Manager Help** menu item has been added to show Task Manager online help information.

Options Menu

The Task Manager **Options** menu contains the following menu items.



Note MULTI remembers the settings of the toggle menu items across debugging sessions.

Item	Meaning
Stop on Task Creation	Specifies whether you want the target's operating system to halt each task as soon as it is created by the application.
Attach on Task Creation	Specifies whether you want to debug the newly created task. If enabled, each newly created task is halted and opened in the Debugger window. This option is applicable only when the Stop on Task Creation menu item is selected.
Print	Prints the tasks currently listed in the Task Manager.
Close	Closes the Task Manager.

View Menu

The Task Manager **View** menu contains the following menu items.

Item	Meaning
Expand Selected Entries	Recursively expands all selected entries in the task list hierarchy.
Expand All Entries	Recursively expands all entries in the task list hierarchy.
Collapse Selected Entries	Collapses all selected entries in the task list hierarchy.
Collapse All Entries	Collapses all entries in the task list hierarchy.
Freeze Task List	Freezes or unfreezes the task list display. For more information, see “Freezing the Task Manager’s Task List Display” on page 427.
Flat View	Changes how tasks are displayed. When Flat View is selected, the tasks are combined together in a single list, regardless of whether they belong to the same processor or high-level RTOS object. When Flat View is cleared, tasks are organized according to processor or high-level object. For more information, see “Task List Pane” on page 440.
Visualize Fine Groups	Toggles the display of an asterisk (*) on Task Manager tabs that correspond to fine task groups. For more information, see “Synchronous Operations” on page 432.
Show AddressSpace IDs	Toggles the display of Address Space IDs in the second column of the Task Manager.
OSA Explorer	Opens the OSA Explorer for the RTOS running on the target. For more information, see “The OSA Explorer” on page 445. This menu item is only available for some RTOSes.
Group Breakpoints	Opens the Breakpoints window for group breakpoints set on the current connection. For reference information about the Breakpoints window, see “The Breakpoints Window” on page 123.

Group Menu

The Task Manager **Group** menu contains the following menu items.

Item	Meaning
Create New Group	Creates a new task group.
Delete Existing Group	Deletes an existing task group.
Add Selected Tasks into Another Group	Adds the selected tasks in the current task group into another task group.
Delete Selected Tasks	Removes the selected tasks from the current task group. You cannot delete tasks from the default task group All .
Continue Selected Tasks	Resumes the selected tasks in the current task group.
Halt Selected Tasks	Halts the selected tasks in the current task group.
Single Step (into function) Selected Tasks	Single-steps the selected tasks in the current task group, stepping into functions.
Single Step (over function) Selected Tasks	Single-steps the selected tasks in the current task group, stepping over functions.
Kill Selected Tasks	Kills the selected tasks in the current task group.
Attach to Selected Tasks	Attaches to the selected tasks in the current task group.
Detach from Selected Tasks	Detaches from the selected tasks in the current task group.
Continue Tasks in Current Group	Resumes all tasks in the current task group.
Halt Tasks in Current Group	Halts all tasks in the current task group.
Kill Tasks in Current Group	Kills all tasks in the current task group.
Attach to Tasks in Current Group	Attaches to all tasks in the current task group.

Item	Meaning
Detach from Tasks in Current Group	Detaches from all tasks in the current task group.
Halt System	Halts all tasks on the target system, thereby freezing the target. This option is not supported on all operating systems.
Run System	Restores the tasks on the target to the status they held before the system was halted.

For more information about task groups, refer to “Working with Task Groups in the Task Manager” on page 428.

Toolbar

The Task Manager toolbar contains the following buttons.

Button	Meaning
	Single-steps the selected tasks in the current task group, stepping into functions.
	Single-steps the selected tasks in the current task group, stepping over functions.
	Resumes the selected tasks in the current task group.
	Halts the selected tasks in the current task group.
	Kills the selected tasks in the current task group.
	Attaches to the selected tasks in the current task group.
	Detaches from the selected tasks in the current task group.
	Adds the selected tasks in the current task group into another task group.
	Removes the selected tasks from the current task group. You cannot delete tasks from the default task group All .
	Freezes or unfreezes the display of the Task Manager window. When the display is frozen (the button is down), a message appears in the status bar at the bottom of the window. For more information, see “Freezing the Task Manager’s Task List Display” on page 427.
	Disconnects from the target.

Button	Meaning
	Opens a Breakpoints window for the group breakpoints set on the current connection. See “Viewing Breakpoint and Tracepoint Information” on page 105.
	Opens the current task’s program in the Project Manager. If there is no current task, the default project opens in the Project Manager.
	Dumps the event log and displays events in the MULTI EventAnalyzer. This button is only available for some RTOSes.
	Opens the OSA Explorer on the RTOS running on the target. For more information, see “The OSA Explorer” on page 445. This button is only available for some RTOSes.
	Prints the tasks currently listed in the Task Manager.

Task List Pane

The area below the toolbar is the task list pane, where the tasks and other kinds of objects (such as CPU nodes) are listed. The columns that are displayed for each task vary according to your operating system. For example, a column might be labeled **Thread** or **Process** depending on the terminology used by the operating system.

By default, the tasks are listed in a hierarchical structure. Tasks are the leaf nodes in the hierarchies, and the non-leaf nodes represent more general concepts in the corresponding RTOS (for example, Node(CPU) and AddressSpace on INTEGRITY). To flatten the hierarchy and list all tasks in one list, choose **View** → **Flat View**. (This menu item is not available for customized task groups.)

The non-leaf nodes are color-coded according to MULTI's color settings. They are displayed as follows:

- The first-level nodes (from the top of the hierarchy) are displayed in the color for strings.
- The second-level nodes are displayed in the color for characters.
- The third-level nodes are displayed in the color for integers.

Task information is displayed in the (foreground) text color. Whenever a task is attached to a Debugger, the corresponding entry in the task list pane has the same background color as the corresponding Debugger window.

Debugging-related statuses in the **Status** column are colored so that you can easily identify them. RTOS statuses are not colored. Many of the statuses that appear in the Task Manager also appear in the Debugger target list. For more information, see “The Status Column” on page 37.

Each time the Task Manager is opened, MULTI automatically displays the tasks with the hierarchies expanded. After that, you control how the hierarchies are displayed. Whenever the Task Manager window is refreshed, MULTI automatically resizes the columns so that all text is visible. However, once you manually change a column's width, MULTI no longer resizes the columns when it refreshes the display.

MULTI maintains the changes you make to the Task Manager window (such as its position) for the current and future debugging sessions.

Information Pane

The area below the task list pane is the information pane, in which the MULTI command pane, target pane, I/O pane, serial terminal pane, Python pane, or traffic pane can be displayed.

By default, the target pane is shown in the information pane area, but you can switch to the MULTI command pane, I/O pane, serial terminal pane, Python pane, or traffic pane by clicking the corresponding tab at the bottom of the Task Manager window.

Tab	Meaning
Cmd or Cmd*	Shows the MULTI command pane in the information pane area.
Trg or Trg*	Shows the target pane in the information pane area.
I/O or I/O*	Shows the I/O pane in the information pane area.
Srl or Srl*	Shows the serial terminal pane in the information pane area.
Py or Py*	Shows the Python pane in the information pane area.
Tfc or Tfc*	Shows the traffic pane in the information pane area.

As in the MULTI Debugger window, an asterisk (*) at the end of a tab name indicates that new messages have arrived in the corresponding pane.

You can change the height of the information pane by dragging the bar that separates it from the task list pane. To automatically resize the panes so that they share the vertical window space evenly, double-click the separator bar.

The Task Manager Shortcut Menu

You can use the mouse to perform useful operations on selected objects, such as expanding or contracting a node and attaching to tasks. Right-clicking in the task list pane opens a shortcut menu with the following items, which perform the same functions as the corresponding items in the menu and toolbar. The appearance of certain menu items is dependent upon your debug server.

Item	Meaning
Stop on Task Creation	Specifies whether you want the target's operating system to halt each task as soon as it is created by the application.
Debug on Task Creation	Specifies whether you want to debug newly created tasks. If selected, each newly created task is halted and displayed in a Debugger window. This option is applicable only when Stop on Task Creation is also selected.
Freeze Task List	Freezes or unfreezes the task list display. For more information, see "Freezing the Task Manager's Task List Display" on page 427.

Item	Meaning
Flat View	Changes how tasks are displayed. When Flat View is selected, the tasks are combined together in a single list, regardless of whether they belong to the same processor or high-level RTOS object. When Flat View is cleared, tasks are organized according to processor or high-level object. This menu item is not available for customized task groups. For more information, see “Task List Pane” on page 440.
Expand Selected Entries	Expands the entire sub-tree for all selected entries. This has no effect on the task entries because they have no sub-trees.
Attach to Selected Tasks	Attaches to the selected tasks. This has no effect on non-task entries.
Detach from Selected Tasks	Detaches from the selected tasks.
Halt Selected Tasks	Halts the selected tasks.
Continue Selected Tasks	Resumes the selected tasks.
Step Selected Tasks	Single-steps the selected tasks.
Detach from Tasks in Current Group	Detaches from all attached tasks in the current task group.
Halt Tasks in Current Group	Halts all tasks in the current task group.
Continue Tasks in Current Group	Resumes all tasks in the current task group.
Step Tasks in Current Group	Single-steps through all tasks in the current task group.
Halt System	Halts all tasks on the target system, thereby freezing the target. This option is not supported on all operating systems.
Run System	Restores the tasks on the target to the status they held before the system was halted.
Add Selected Tasks into Group	Adds the selected tasks into another task group.
Delete Selected Tasks from Group	Removes the selected tasks from the current task group.

Item	Meaning
Show AddressSpace IDs	Displays Address Space IDs in the second column of the Task Manager.
Other entries	Displays or hides the corresponding column.



Note In the shortcut menu, the word `Task` is replaced with the corresponding terminology used by the underlying RTOS (such as `Process` or `Thread`).

Task Group Configuration File

The definitions for task groups are persistent across debugging sessions. These definitions are stored in a configuration file. When launching MULTI from the command line, use the `-tv` option to specify which configuration file is to be used in the debugging session. For example:

```
-tv mytaskview.tvc
```

If no file extension is given in the task view configuration filename, `.tvc` is appended to the filename. If no configuration file is specified from the command line, `taskview.tvc` from your local configuration directory is used.

For each group, MULTI keeps a task fingerprint for each task in the group. Whenever the group is to be displayed (when you select the corresponding tab in the Task Manager), MULTI uses this task fingerprint to determine which tasks to display. Considering different debugging environments, MULTI provides the following three criteria to match a task against a task fingerprint.

- **Task Id + Hierarchy** — When a task's identifier (a number) and the hierarchy path are the same as those kept in a task fingerprint of a group, the task is considered to be in the group.
- **Task Name + Hierarchy** — When a task's name and the hierarchy path are the same as those kept in a task fingerprint of a group, the task is considered to be in the group.
- **Task Id or Name + Hierarchy** — When a task's name or identifier and the hierarchy path are the same as those kept in a task fingerprint of a group, the task is considered to be in the group.

See also the description of the **TaskMatchCriteria** configuration option in “The More Debugger Options Dialog” in Chapter 9, “Configuration Options” in the *MULTI: Editing Files and Configuring the IDE* book.

MULTI also provides a configuration option **DeleteDeadTaskFromGroup** to tell the Task Manager to delete *dead* task fingerprints from the task group. When you display a task group by clicking the corresponding tab, if there is no live task for a task fingerprint, the task fingerprint is *dead*. For example, a group created in the last debugging session contains a fingerprint for `task1`, but `task1` is not on the task list at present because the task has not yet been created by the program in the current debugging session. If the configuration is on when you click the tab for the group, the Task Manager permanently removes the fingerprint for `task1` from the group. If this option is off, the fingerprint for `task1` is not removed and `task1` shows up when it is created. However, the fingerprints for zombied tasks are not cleaned up from the group. (For more information about the **DeleteDeadTaskFromGroup** option, see “The More Debugger Options Dialog” in Chapter 9, “Configuration Options” in the *MULTI: Editing Files and Configuring the IDE* book.)



Note The configuration file contains the information about group definitions and synchronous operations on tasks.

OS-Awareness in Run Mode

Two OS-aware tools allow you to browse OS information while debugging in run mode: the **OSA Explorer** and the OSA Object Viewer. The **OSA Explorer** is useful for browsing information about an entire operating system, while the OSA Object Viewer is useful for browsing information about an individual INTEGRITY object. The following sections briefly describe each tool.

The OSA Explorer

The **OSA Explorer** shows INTEGRITY operating system tasks and other objects on the target in run mode. You can launch an **OSA Explorer** anytime; however, if the system is not halted, doing so causes MULTI to halt the system. In this case, closing the **OSA Explorer** causes MULTI to resume system execution. For more information about **System** halts, see “Default Task Groups” on page 428.

To launch an **OSA Explorer** in run mode, do one of the following:

- In the Task Manager or Debugger, click the **OSA Explorer** button (.
- In the Task Manager or Debugger, select **View** → **OSA Explorer**.
- In the Debugger command pane, enter the **osaexplorer** command. For information about this command, see “Object Structure Awareness (OSA) Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

For more information about the **OSA Explorer**, see Chapter 19, “Freeze-Mode Debugging and OS-Awareness” and “Object Structure Aware Debugging” in the *INTEGRITY Development Guide*.

The OSA Object Viewer

With the OSA Object Viewer, you can view INTEGRITY object attributes, inject messages into the kernel, and navigate between objects. The OSA Object Viewer also keeps a history of all the objects you have viewed. You can easily navigate this history. Because the OSA Object Viewer is typically used to display small chunks of information from the kernel, it is faster than the **OSA Explorer**.



Note The OSA Object Viewer is only available if you are debugging a run-mode connection and using INTEGRITY version 10 or later.

To open the OSA Object Viewer, do one of the following:

- In the Debugger target list, select an object and click the **OSA Object Viewer** button (.
- In the Debugger command pane, enter the **osaview** command. For information about the **osaview** command, see “Object Structure Awareness (OSA) Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.
- In the Debugger source pane, double-click an INTEGRITY object variable while the task is halted.

The OSA Object Viewer opens on an object summary, a list of attributes (under the **Attributes** tab), and a list of operations you can perform (under the **Operations** tab). Objects that are underlined in the OSA Object Viewer

are linked to more detailed information. To view this information, click the underlined object. Similarly, clicking an underlined operation performs the operation.

By default, the OSA Object Viewer window is reused. However, if you freeze the window or click the **Reuse** button () so that it no longer appears to be pushed down, a new window appears the next time you open an OSA Object Viewer or the next time you click a linked (that is, underlined) object in the OSA Object Viewer.

For more information about the OSA Viewer, see “Object Structure Aware Debugging” in the *INTEGRITY Development Guide*.

Profiling All Tasks in Your System

If you are using INTEGRITY version 10 or later, MULTI allows you to view processor usage for all tasks in your system. The information is updated continually while the target is running, and it can be obtained without instrumenting the code: INTEGRITY periodically samples the program counter (PC) of running tasks, and the run-mode debug server (**rtserv2**) delivers the resulting PC samples to MULTI.

The continually updated profiling information is displayed both in the **Profile** window’s standard calls and source reports (see “Standard Calls Report” on page 352 and “Source Report” on page 356) and in a target list column labeled **CPU %**. This column displays the percentage of the CPU that each task and AddressSpace is currently using. The AddressSpace **CPU %** is simply the sum of the CPU percentages for all the tasks in that AddressSpace.

To begin profiling all tasks in your system, perform the following steps:

1. In the Debugger’s target list, select the run-mode connection to your target.
2. Open the **Profile** window by selecting **View → Profile** or by entering the **profile** command. For information about the **Profile** window, see “The Profile Window” on page 348. For information about the **profile** command, see Chapter 13, “Profiling Command Reference” in the *MULTI: Debugging Command Reference* book.



Tip You can use the **ServerPollinterval** configuration option to control the interval between updates to displayed data. For more information about this option, see “The More Debugger Options Dialog” in Chapter 9, “Configuration Options” in the *MULTI: Editing Files and Configuring the IDE* book.

To disable profiling, close the **Profile** window.

Freeze-Mode Debugging and OS-Awareness

Chapter 19

This Chapter Contains:

- Starting MULTI for Freeze-Mode Debugging and OS-Awareness
- The OSA Explorer
- Debugging in Freeze Mode
- Configuration File

When you debug a multitasking application, MULTI interacts with the target's operating system in freeze mode, in run mode, or in both freeze and run mode simultaneously. In freeze mode, the entire target system stops when a breakpoint is hit or a task is halted. In run mode, the operating system kernel continues to run as you halt and examine individual tasks. For information about run-mode debugging, see Chapter 18, "Run-Mode Debugging". For information about debugging in run mode and freeze mode, see "Freeze-Mode and Run-Mode Connections to the Same Target" on page 70.

During freeze-mode debugging, MULTI allows you to debug individual program tasks, and it provides OS-awareness for operating system objects, allowing you to examine detailed information about these objects. With freeze-mode debugging, you can:

- Display an **OSA Explorer** showing operating system tasks and other objects on the target (if any).
- Display an entry in the target list for each task, even if that task is suspended or otherwise not running.
- Set task-specific breakpoints that will stop the running process only when the chosen task is currently executing.
- Display call stacks and stack (local) variables for any task that can be debugged.
- Perform task-specific single-stepping.



Note The **OSA Explorer** is also available in some run-mode debugging environments. For more information, see "The OSA Explorer" on page 445.

Currently, freeze-mode debugging is available for these target environments:

- INTEGRITY
- velOSity
- u-velOSity
- ThreadX

If your operating system is not on the list, contact your Green Hills sales representative.



Note In this chapter, *task* is used as a general term to describe the target operating system's unit of program execution. Specific terminology varies according to the target's operating system (for example, a task may also be called a *thread*). When the term *process* is used in this chapter, it usually refers to the master process. For more information, see "Master Debugger Mode" on page 458.

Starting MULTI for Freeze-Mode Debugging and OS-Awareness

MULTI identifies the operating system in use on your target and then configures itself to debug that operating system. INTEGRITY and velOSity are identified by OSA debugging information in the operating system kernel. MULTI identifies other operating systems by examining programs for the following OS-specific symbols:

- u-velOSity — `uv_task_create` or `_uv_task_create`
- ThreadX — `_tx_thread_created_ptr`

Under special circumstances, an OSA integration package may be provided to allow freeze-mode OS-aware debugging. In that case, start MULTI with the following command line option:

```
-osa osa_name[#cfg=configuration_file][#lib=library_name][#log=log_file]
```

where:

- *osa_name* is case-insensitive.
- *configuration_file* is the configuration file for freeze-mode debugging and OS-awareness (see "Configuration File" on page 468). If you do not specify a full path to the configuration file, MULTI first searches for it in the **os_aware** directory of your personal MULTI configuration directory, and then looks for it in the MULTI installation directory. If you do not specify a configuration file, *osa_name.osa* is used by default.
- *library_name* is the library containing the OSA integration package (see "The OSA Integration Module Name" on page 471). The library may be a DLL file (Windows) or an SO shared library (UNIX). If you do not specify a library, *osa_name.dll* (Windows) or *osa_name.so* (UNIX) is used by default.

You can place the integration module in any location, and use an absolute or relative path to locate it from MULTI's command line or from the configuration file. If the integration module is specified with only a base

name, MULTI first searches for it in the MULTI installation directory, and then in a way defined by the host machine.

- *log_file* is the name of the file in which you want to log the interaction between MULTI and the OSA integration package. This option is seldom required.

For example, to debug a program **my_program** in freeze mode with an OSA integration package called *rtos*, which uses a DLL called **rtos.dll**, start MULTI with the following command line option:

```
multi -osa rtos#lib=rtos_lib#log=rtos_logfile my_program
```

If you are debugging multiple programs and you want to use OS-aware debugging features for some of the programs, but not for others, set the option `RequestOsaPackage` to `On`. With this option, you do not have to specify the **-osa** option on the command line. Instead, MULTI prompts you for the OS-awareness package name when you are debugging in freeze mode. MULTI resolves the name of the configuration file and of the library as described above. See also “The More Debugger Options Dialog” in Chapter 9, “Configuration Options” in the *MULTI: Editing Files and Configuring the IDE* book.



Tip You can also use the **osasetup** command to specify an OSA integration package. For information about this command, see “Object Structure Awareness (OSA) Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

The OSA Explorer

The **OSA Explorer** shows operating system tasks and other objects on the target (if any). The following sections explain how to display and customize the **OSA Explorer** and describe the operations that can be performed in the kernel object list.

Displaying an OSA Explorer

When the target is halted and you are debugging a multitasking application with OS-awareness, you can launch an **OSA Explorer** by doing one of the following:

- In the Debugger, select **View → OSA Explorer**.

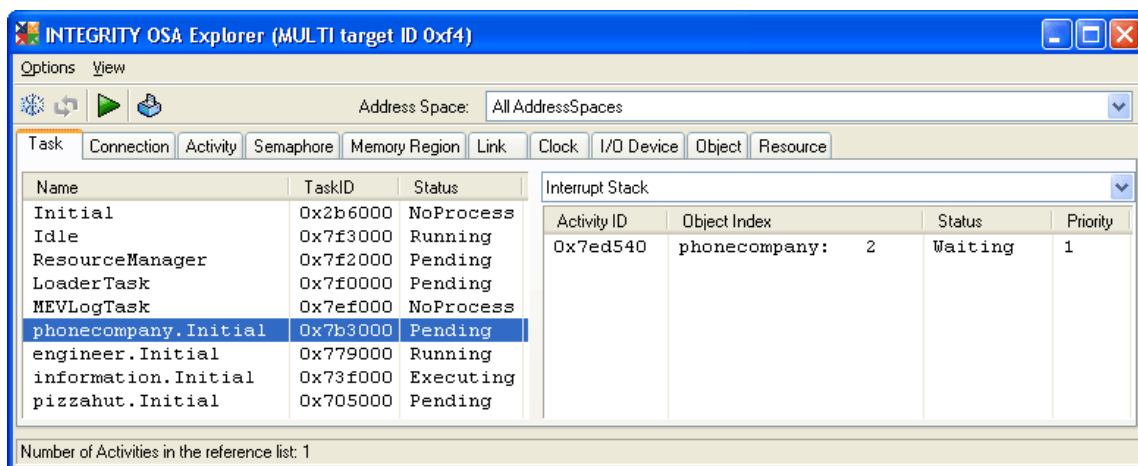
- In the Debugger command pane, enter the **osaexplorer** command. For information about this command, see “Object Structure Awareness (OSA) Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

If the **View → OSA Explorer** menu item is dimmed out or the **osaexplorer** command prints an error such as:

```
osaexplorer: not supported in the current environment
```

then MULTI does not recognize the target operating system as one for which MULTI can perform freeze-mode OS-aware debugging. For more information, see “Starting MULTI for Freeze-Mode Debugging and OS-Awareness” on page 451.

The following is an **OSA Explorer** showing a task list of all AddressSpaces for the INTEGRITY operating system.



The following is the same **OSA Explorer** showing the MemoryRegion (a type of kernel object) list for the AddressSpace engineer on the INTEGRITY operating system.

Memory Region ID	Address Space	Beginning	End	Physical Beginning	Physical End	Object Index
0x7cd500	engineer	0x137000	0xffffffff	Not Mapped	Not Mapped	kernel: 129
0x7cd580	engineer	0x10000	0x10fff	0x7b1000	0x7b1fff	kernel: 131
0x7cd600	engineer	0x11000	0x11fff	0x7ae000	0x7aefff	kernel: 133
0x7cd680	engineer	0x12000	0x12fff	0x7ad000	0x7adfff	kernel: 135
0x7cd700	engineer	0x13000	0x13fff	0x7ac000	0x7acfff	kernel: 137
0x7cd780	engineer	0x14000	0x14fff	0x7ab000	0x7abfff	kernel: 139
0x7cd800	engineer	0x15000	0x15fff	0x7aa000	0x7aafff	kernel: 141
0x7cd880	engineer	0x16000	0x16fff	0x7a9000	0x7a9fff	kernel: 143
0x7cd900	engineer	0x17000	0x17fff	0x7a8000	0x7a8fff	kernel: 145
0x7cd980	engineer	0x18000	0x18fff	0x7a7000	0x7a7fff	kernel: 147
0x7cda00	engineer	0x19000	0x19fff	0x7a6000	0x7a6fff	kernel: 149
0x7cda80	engineer	0x1a000	0x1afff	0x7a5000	0x7a5fff	kernel: 151

Number of Memory Regions: 34

Each type of object that can be explored is represented as a tab in the **OSA Explorer**. If an object type has no reference object, the **OSA Explorer** contains one object list pane to show the instances of this type of object. If an object type has a reference object, two object list panes are displayed:

- The master pane on the left side of the **OSA Explorer**.
- The reference pane on the right side of the **OSA Explorer**.

Whenever an object is selected in the master pane, the instances of the corresponding reference object are shown in the reference pane.

Because the object shown in the master pane could have references to more than one type of object, the drop-down list at the top of the reference pane indicates the object type being shown in the reference pane. If you select an object type to be shown in the reference pane, your change is persistent for the duration of the debugging session.

For information about INTEGRITY objects available in the **OSA Explorer**, see the *INTEGRITY Development Guide*.

The OSA Explorer GUI Reference

The following table describes the menu items and toolbar buttons available in the **OSA Explorer**.

Menu Item	Button Equivalent	Meaning
Options → Print		Prints the object list(s) currently displayed in the OSA Explorer . If two object lists (the master list and the reference list) are displayed in the OSA Explorer , the print dialog box appears twice.
Options → Help	no equivalent button	Displays online help information about freeze-mode debugging and the OSA Explorer .
Options → Close		Closes the OSA Explorer . Whether this button appears on your toolbar depends on the setting of the option Display close (x) buttons . To access this option, select Config → Options → General Tab .
View → Freeze Object List		Freezes or unfreezes the automatic refreshing of the OSA Explorer . When the OSA Explorer is frozen, a message appears in the status bar at the bottom of the window, the contents of the window are preserved, and you cannot switch to another tab. The window is not updated again until it is unfrozen.
View → Manually Refresh Object List		Refreshes the OSA Explorer even if it is frozen. This button is not available if the target is halted.
View → Toggle Target Status		Runs/stops the underlying process on the target if you are debugging in freeze mode, or resumes/halts the underlying RTOS on the target if you are debugging in run mode.

Customizing the Object List

The **OSA Explorer** for each OS has a default configuration that indicates which columns to display and how to display them for each type of object. The object list pane behaves like other multiple-column panes in MULTI. You can customize it as follows:

- To sort a list based on a column's contents, click that column's header. The sorting toggles between ascending and descending order.
- To move a column, drag that column's header left or right.
- To change the width of a column, drag the column boundary.
- To show or hide a column, right-click the pane to open a shortcut menu, and select or clear the corresponding menu items.

Changes that you make to the **OSA Explorer** itself (such as its size and position) and to each object are maintained across debugging sessions.

Task information is displayed in the (foreground) text color. Entries in the object list pane have the same background color as the corresponding Debugger window (only applicable if you have opened multiple Debugger windows).

Object List Operations

When you double-click a task in the object list pane, the selected task loads in the current Debugger window. When you right-click an object in the object list pane, a shortcut menu appears.

The following table describes the menu items that may appear in the **OSA Explorer** shortcut menu. The actual menu contents vary depending on the RTOS and object type.

Menu Item	Meaning
Column Name	Toggles between showing or hiding the indicated column.
Freeze Object List	Freezes or unfreezes the automatic refreshing of the OSA Explorer . When the OSA Explorer is frozen, a message appears in the status bar at the bottom of the window, and the contents of the window are preserved. The window is not updated again until it is unfrozen.
View Object Information	Displays the selected object in a Data Explorer.

Menu Item	Meaning
Debug Task	Displays the selected task in the Debugger window. See “Debugging in Freeze Mode” on page 457 and “Shortcut Menu Entry Definitions” on page 475.
Inject Message	Injects a message to the underlying RTOS to change its behavior. For example, on INTEGRITY, you can dynamically inject a message into the system to: • Take a <code>Semaphore</code> object on behalf of a task • Release a <code>Semaphore</code> to break a dead lock • Send a message to a <code>Connection</code> object This option is only supported on some RTOSes.

To print the contents of the object list pane, select **Options → Print**.

Debugging in Freeze Mode

The following sections explain how to debug in freeze mode:

- “Master Debugger Mode” on page 458
- “Task Debugger Mode” on page 459
- “Task-Specific Single-Stepping” on page 460
- “Working with Freeze-Mode Breakpoints” on page 462
- “Program I/O” on page 463
- “Multi-Core Debugging” on page 464

If the currently executing task resides in the kernel address space, you can either use Master Debugger mode or Task Debugger mode to view information specific to the task. If you want to debug a task that is not currently executing or does not reside in the kernel address space, you must use Task Debugger mode to view task-specific information.



Note The configuration option **OSATaskAutoAttachLimit** allows you to specify the maximum number of OSA tasks that are displayed in the target list (the default is 32). The tasks are displayed consecutively until this limit is reached. For example, if this option is set to 32 and there are 33 tasks, the 33rd task is not displayed in the target list. To manually display additional tasks in

the target list, double-click the tasks in the **OSA Explorer Task** pane. For more information about the configuration option **OSATaskAutoAttachLimit**, see “Other Debugger Configuration Options” in Chapter 9, “Configuration Options” in the *MULTI: Editing Files and Configuring the IDE* book.



Note The configuration option **OSASwitchToUserTaskAutomatically** controls whether the Debugger automatically displays the currently executing user-mode task when the target is stopped while executing user-mode tasks. This is in addition to automatically displaying all of the tasks in the system as controlled by the **OSATaskAutoAttachLimit** configuration option. For more information about these configuration options, see “Other Debugger Configuration Options” in Chapter 9, “Configuration Options” in the *MULTI: Editing Files and Configuring the IDE* book.

Master Debugger Mode

During freeze-mode debugging, selecting the master process in the target list enables Master Debugger mode for the target. The master process is displayed in the target list after the CPU node of a freeze-mode connection and before the OSA address spaces whose names are prefixed with **OSA**. When the master process is selected, the kernel’s source code is displayed in the Debugger window. Double-clicking the process displays the kernel source code in a new Debugger window.

The status of the master process (displayed in the **Status** column) corresponds to the status of a normal process or provides specific information about the mode in which the CPU may be stopped. For example, the status may be **Running**, **Stopped**, **Stopped in Kernel Mode**, or **Stopped in User Mode**. All operations, such as memory access, register access, etc., are performed in the target’s current status. If the master process is selected in the target list while the target is stopped in user mode (that is, in a non-kernel task), the PC pointer becomes gray.

The following operations are only available in Master Debugger mode:

- Reloading or restarting the program.
- Killing the process or detaching from it.
- Setting normal breakpoints (as opposed to task-specific and any-task breakpoints). All breakpoints set in Master Debugger mode are normal breakpoints.
- Setting hardware breakpoints.

When you reload or restart the program or detach from or kill the process, MULTI removes all OSA task entries from the target list and closes the **OSA Explorer** and any Debugger windows that you have opened on an OSA task (by double-clicking the task in the target list or by selecting **Open in New Window** from the target list's right-click menu).

Task Debugger Mode

To view source code for a task, do one of the following:

- In the **OSA Explorer Task** pane, double-click a task.
- In the **OSA Explorer Task** pane, right-click a task and choose **Debug Task**.
- In the Debugger's target list, select a task. Tasks are located under an address space node whose name begins with **OSA**.
- In the Debugger command pane, enter the command **osatask**. For information about this command, see "Object Structure Awareness (OSA) Commands" in Chapter 15, "Scripting Command Reference" in the *MULTI: Debugging Command Reference* book.

To open a separate Debugger window on a task, double-click the task in the target list. MULTI does not open multiple Debugger windows for the same task. If a Debugger window already exists for a double-clicked task, MULTI brings that window to the foreground instead of opening a new window.

The following list describes behavior that is specific to Task Debugger mode:

- Call stacks, local variables, and register displays are specific to the task.
- The title bar displays the task ID (a number that uniquely identifies the task) and the task name, if the OS maintains a name for that task (see "The Object Attribute Definition" on page 474).

- When the RTOS's master process is halted, the target list's **Status** column displays the status of tasks in the RTOS. When the RTOS's master process is running, the **Status** column displays **<OS Running>**.
- Except for the  and  buttons, buttons related to execution (such as , , ) and their corresponding commands are only available when the master process is halted and the task displayed is the currently executing task.
- Single-stepping and breakpoints behave in a manner that is more appropriate to task-aware debugging. For more information, see the next two sections.
- Some of the MULTI commands that affect the global debugging state are not available in Task Debugger mode. If you try to run one of these commands, MULTI displays the following error message:

This command cannot be run with an OSA Task selected.
Select the target list entry corresponding to the connection
or executable first.

Task-Specific Single-Stepping

When you single-step a process in Task Debugger mode, MULTI avoids stopping the process when another task is running. This makes debugging shared code more straightforward by allowing you to step through one task at a time.

MULTI generally single-steps through program source by setting temporary breakpoints and running until one of those breakpoints is hit. Sometimes a context switch takes place between the time the process is run and the time one of the temporary breakpoints is hit. (It is possible for the temporary breakpoint to be hit from another task's context.) In these cases, MULTI lets the target process run until one of the following occurs.

- The temporary breakpoint is hit while the original task is running.
- Another breakpoint is hit.
- The running process is halted.

This feature operates transparently, and no special action is required, aside from performing all task-specific single-steps in Task Debugger mode. In Master Debugger mode, single-stepping may occasionally step over context switches.

The following example helps to illustrate when task-specific single-stepping can be useful. This example uses u-velOSity's GH-API, but most other operating systems can perform similar operations.

```

335 1    void shared_func(char *s) {
336 2        printf("%s called shared_func\n",s);
-> 337 3        gh_task_yield();
338 4        printf("%s end of call to shared_func\n",s);
339 5    }
340
341 1    void task_1_entry(GH_ADDRESS input) {
342 2        for(;;)
343 3            shared_func("task_1");
344 4    }
345
346 1    void task_2_entry(GH_ADDRESS input) {
347 2        for(;;)
348 3            shared_func("task_2");
349 4    }

```

In this process, two tasks are running at the same priority without time slicing. Both tasks loop, calling `shared_func()`. The first task, `task_1`, has `task_1_entry()` as its entry point. The second task, `task_2`, has `task_2_entry()` as its entry point. Within `shared_func()`, each task calls `gh_task_yield()`, a GH-API call that allows other tasks at the same priority to run. In this case, `task_1` yields to `task_2` and then `task_2` yields to `task_1`. So in the steady state, one task is always suspended within a call to `gh_task_yield()` while the other task is running.

At the moment the process is stopped, `task_1` is the current task and is stopped on line 337, where it is just about to call `gh_task_yield()`. If you now press the  button from within `task_1`'s Debugger window, the following happens:

1. MULTI sets a temporary breakpoint on line 338 and restarts the process.
2. The call to `gh_task_yield()` causes `task_1` to suspend and `task_2` to run again.
3. `Task_2` returns from its previously suspended call to `gh_task_yield()` and the process hits the temporary breakpoint.
4. MULTI notices that `task_1` is not the current task and restarts the process again.

5. `Task_2` loops and calls `gh_task_yield()`, which causes `task_2` to suspend and `task_1` to run again.
6. `Task_1` returns from its call to `gh_task_yield()` and the process again hits the temporary breakpoint.
7. MULTI notices that `task_1` is the current task and stops the process.

Task-specific single-stepping makes it easy for you to concentrate on debugging the task you are interested in.

Working with Freeze-Mode Breakpoints

When performing freeze-mode debugging on multitasking applications, you can set task-specific breakpoints, any-task breakpoints, and normal breakpoints. Each type of breakpoint is described below:

-  — Indicates a task-specific breakpoint. A task-specific breakpoint can be set in an OSA task and can only be hit by the task it was set on. A task-specific breakpoint is implemented as a conditional breakpoint: the current task ID when the breakpoint is hit must match the task ID associated with the breakpoint. Otherwise, MULTI simply continues running the process.

If the breakpoint icon is red, the breakpoint can only be hit by the task currently selected in the target list (that is, the breakpoint was set on the current task). If the breakpoint icon is gray, the breakpoint can only be hit by a task other than the one currently selected in the target list (that is, the breakpoint was *not* set on the current task). Task-specific breakpoints in the kernel address space are gray in the OSA master process.

In the **Breakpoints** window, task-specific breakpoints are listed with a command parameter. Otherwise they are displayed like normal breakpoints.

-  — Indicates an any-task breakpoint. An any-task breakpoint can be set in an OSA task and can be hit by any task in the same address space.

To set an any-task breakpoint, select a task in the target list and hold the Shift key as you click a breakdot (•) in the Debugger window. Alternatively, you can use the **sb at** command.

For example:

```
sb at function#5
```


sets an any-task breakpoint at line 5 of the function *function*. For information about the **sb** command, see Chapter 4, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

-  and all other breakpoint icons — Represent normal breakpoints. A normal breakpoint can be set in the OSA master process and can be hit by OSA tasks that can access the breakpoint’s address.

These breakpoints are similar to any-task breakpoints set in OSA tasks, but they are not the same. You cannot set an any-task breakpoint in the OSA master process. The freeze-mode debug server has no knowledge of RTOS concepts such as any-task breakpoints; it treats the program you are debugging as a stand-alone program. MULTI uses the OSA package to simulate an RTOS scenario (RTOS concepts are only available to MULTI). MULTI translates the semantics of RTOS operations into the normal operations supported by the debug server.

You can only set hardware breakpoints in the OSA master process. These breakpoints are not RTOS aware, so they will affect all address spaces.



Note Only one breakpoint can be set at a particular address. For example, if a task-specific breakpoint is set at the location `shared_func#2()`, the breakpoint must be removed before another breakpoint (of any type) can be set at the same address. If you do not remove the breakpoint, MULTI does so for you. On INTEGRITY, this limitation applies to each AddressSpace (that is, multiple breakpoints can be set at the same address as long as each breakpoint is set in a different AddressSpace).

Program I/O

In a freeze-mode environment, program I/O is usually redirected to the MULTI Debugger and immediately displayed in its **I/O** and **Cmd** panes. Operating systems such as INTEGRITY, however, capture and independently handle program I/O. In this case, I/O may not be immediately visible.

When an INTEGRITY program calls an I/O function such as `fprintf()` to print a message, input is neither immediately sent nor output immediately received. Instead, INTEGRITY translates the request into a console operation and queues it in the system. Even a flush operation, such as `fflush()`, is translated and queued in the same way. When the kernel is running and handling these operations, I/O is printed to the console approximately every second.

When you single-step on a task in freeze mode, you cannot immediately see the output of I/O functions because the kernel is halted. The output is displayed in the console only after you start the target running and the kernel has a chance to flush out queued console operations.

Multi-Core Debugging

A *multi-core target* is a target containing multiple processors, or *cores*, that can each be debugged independently. This section details techniques and limitations specific to debugging multi-core targets.

Multiple Cores in the Target List

When you connect to a multi-core target, MULTI displays the cores as separate entries in the target list, and it assigns each core a number, beginning with zero.

Using Hook Commands in Multi-Core Setup Scripts

Before loading a program onto a hardware target, MULTI typically runs the board setup script (if any) associated with the Connection Method. The board setup script may set up initial register values, memory controller configurations, etc. Board setup scripts for multi-core targets can take advantage of the hook commands documented in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

For example, suppose you want to run several MULTI commands on core 1 after resetting your target. You might add a command such as the following to your setup script:

```
addhook -core 1 -after reset {commands}
```

If you are using the hook commands, you may want your setup script to run once upon connection rather than before every download. For more information, see “Early MULTI Board Setup Scripts with Debugger Hooks” in Chapter 2, “Setting Up Your Target Hardware” in the *MULTI: Configuring Connections* book.

Preparing Multiple Cores to Run a Single Executable

If multiple cores on your target share memory, or if one core controls the reset and/or run-control capabilities of others, you might choose to build the code for all the cores into a single executable file. Before loading this executable onto your target, you must associate the executable with every core. One method for doing so follows:

1. Select the executable in the target list, and click the **Connect** button ().
2. Using the **Connection Chooser** that appears, connect to your multi-core target.
3. In the **Use Which Connection/CPU?** dialog box that appears, select any one of the cores, and click **OK**.
4. In the Debugger command pane, issue the **new** command with no arguments. For information about the **new** command, see Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.
5. If the **Use Which Connection/CPU?** dialog box appears again, select any core. If the dialog box does not appear, MULTI automatically associated the executable with the only remaining core.
6. Repeat steps 4 and 5 until the executable has been associated with every core on the target.

For information about the **Use Which Connection/CPU?** dialog box, see “Associating Your Executable with a Connection” on page 87.

After associating the executable with every core on the target, you can load the executable onto the target:

1. In the target list, select the executable listed after the core that is supposed to run first.
2. In the Debugger command pane, issue the **prepare_target -load** command. For information about this command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.
3. For each remaining core, select the executable listed after the core, and enter **prepare_target -verify=none** in the Debugger command pane.

Alternatively, you can use the **prepareAllCores** option to configure MULTI to automatically and silently run **prepare_target -verify=none** on all secondary cores. For more information, see the description of **prepareAllCores** in “Other Debugger Configuration Options” in Chapter 9, “Configuration Options” in the *MULTI: Editing Files and Configuring the IDE* book.

Preparing Multiple Cores to Run Separate Executables

If the cores on your target do not share memory and if each can be independently controlled, or if they are otherwise intended to execute programs independently, you might choose to build the code for each core into its own separate executable file. Before loading the executables onto your target, you must associate each executable with the core that is supposed to run it. One method for doing so follows:

1. Select one of the executables in the target list, and click the **Connect** button ().
2. Using the **Connection Chooser** that appears, connect to your multi-core target.
3. In the **Use Which Connection/CPU?** dialog box that appears, select the core that you want to run the selected executable, and click **OK**.
4. In the Debugger command pane, enter **new program_name**, where *program_name* is the name of another executable. For information about the **new** command, see Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.
5. If the **Use Which Connection/CPU?** dialog box appears again, select the core that you want to run the second executable. If the dialog box does not appear, MULTI automatically associated the executable with the only remaining core.
6. Repeat steps 4 and 5 until an executable has been associated with every core on the target.

For information about the **Use Which Connection/CPU?** dialog box, see “Associating Your Executable with a Connection” on page 87.

After associating each executable with the core that is supposed to run it, you must separately load each program onto the corresponding core:

1. Select an executable in the target list.
2. In the Debugger command pane, issue the **prepare_target -load** command. For information about this command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.
3. Select a different executable in the target list.
4. If your board setup script only affects the current core when it is run, and if it does not reinitialize any shared components on the board (such as memory) — Issue the **prepare_target -load** command again.

If your board setup script affects multiple cores when it is run, or if it reinitializes shared components on the board (such as memory) — Issue the **load -nosetup** command. For information about this command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.

5. Repeat steps 3 and 4 for each remaining executable.

Synchronous Run Control

Some targets support (or require) *synchronous*, or simultaneous, run-control operations on all cores. To initiate synchronous run-control operations, you must do one of the following:

- Use the options in the Task Manager’s **Group** menu (for example, **Continue Tasks in Current Group**).
- Open the Task Manager, and then issue the **groupaction** command with appropriate arguments:

groupaction -r|-h|-s @All

where **-r**, **-h**, or **-s** specifies a run, halt, or single-step operation, respectively, and **@All** indicates that the designated operation be synchronously performed on all cores on the target. In a freeze-mode environment, synchronous run control is not supported on groups other than **All**.



Note Selecting multiple tasks in the target list and then issuing a MULTI command such as **c** or **halt** does *not* initiate synchronous run-control operations.

If you are using a Green Hills Probe, see the *Green Hills Debug Probes User's Guide* for any additional configuration settings that apply to your target.

Limitations

- If multiple cores on your target share memory and a single executable, then breakpoints, command line procedure calls, and host I/O system calls are only supported on the core that you initially loaded the program onto. Many advanced debugging features such as coverage, call count, and call graph profiling make use of breakpoints, command line procedure calls, and host I/O and are therefore also only supported on the first core.
- On targets that require synchronous run control, conditional breakpoints and breakpoints that resume the target are not supported.
- If one core holds another core in a reset state on your target, your debugging interface (for example, Green Hills Probe) may not be able to manipulate the latter core until the former brings it out of reset. In this case, the Debugger may show the core held in reset as **Running**, even though it is not actually executing any code. You may not be able to halt a core that is being held in reset.

Configuration File

The configuration file for freeze-mode debugging and OS-awareness provides the Debugger with information about objects to be explored. For example, the file may:

- Assign identification numbers to each object and its attributes.
- Specify the relationships between objects.
- Define how the attributes of an object are to be displayed in the **OSA Explorer**.

The name of the configuration file consists of the lowercase name of the corresponding operating system or of your OS-awareness package. The filename ends with a **.osa** extension. It is stored in the directory **os_aware** under

your personal MULTI configuration directory or in the following directory under the MULTI installation directory:

- Windows — **defaults\os_aware**
- UNIX — **defaults/os_aware**

MULTI searches for the configuration file in your personal MULTI configuration directory first, and then looks for it in the MULTI installation directory.

In order to make the configuration file more readable, comment lines are supported. If the first two non-space characters are forward slashes (“//”), then the line is considered to be a comment.



Note Two consecutive forward slashes (“//”) in the middle of a line is not considered to be a comment indicator as it would be in C++ code.

Each line in the configuration file can be a comment line, an empty line, or a line representing a configuration element. A backslash (`\`) at the end of a line combines the next line with it, so multiple lines can be combined together to represent a single configuration element.

Even though MULTI recognizes the reserved words in a case-insensitive way, we recommend keeping all reserved words in uppercase for readability purposes as we do in the configuration files for the built-in OS integrations.

All syntax elements of the configuration file are separated by a colon (':'). If a string provided by the OSA integration provider contains a colon (':'), the string should be enclosed in double quotes.

The design of MULTI's OSA integration mechanism totally separates the GUI representation from program logic. Each object, attribute of an object and reference of an object is assigned a unique non-negative number. Negative numbers are reserved for special purposes. In the interaction between MULTI and the OSA integration module, only the identifier numbers are used. This makes it very easy for you to customize the strings displayed in an **OSA Explorer**, including internationalization.

The following subsections specify the syntax of the elements in a configuration file.

General Settings

The OSA Integration Version

Format:

OSA_CONFIG:OSA_VERSION:*version_number*

version_number is an integer. The current version number is 2.

Terminology for Tasks

Format:

OSA_CONFIG:OSA_TASK_ALIAS:*terminology_for_task*

The *terminology_for_task* specified here will be used at some places in the **OSA Explorer**. If the setting is absent in the configuration file, `task` will be used as the default.

When a task name needs to be used in other places of the configuration file, use `OSA_TASK` instead of *terminology_for_task*.

The Menu Name Shown in the Debugger

Format:

OSA_CONFIG:OSA_MULTI_MENU:*menu_name*

The *menu_name* argument should be a constant string, and must not contain “&”. It is not used in the current MULTI implementation.

The OSA Explorer Title

Format:

OSA_CONFIG:OSA_EXPLORER_TITLE:*osa_explorer_title*

The *osa_explorer_title* argument specifies the title for the **OSA Explorer**. It is a string that can contain a number replacement (for target ID used by MULTI) in the syntax of `printf()` in the C library (`%d` or `0x%x`).

Initial OSA Commands

Format:

OSA_CONFIG:OSA_INIT_COMMAND:*initial_osa_commands*

When MULTI initializes the OSA integration module, *initial_osa_commands* are sent to it immediately.

Multiple initialization commands can be specified on multiple lines. When the module is initialized, MULTI sends the commands to the OSA integration module sequentially. If an initialization command fails, MULTI ignores the rest of the initialization commands, but continues as if the OSA integration module had been successfully initialized.

The Poll Interval

Format:

OSA_CONFIG:OSA_POLL_INTERVAL:*interval_in_milliseconds*

The OSA integration module can provide a function to be called periodically by MULTI, with an interval of *interval_in_milliseconds*. If the setting is absent, the default value (200 milliseconds) will be used.

The OSA Integration Module Name

Format:

OSA_CONFIG:OSA_MODULE_NAME:*osa_integration_module_name*

With the exception of the built-in integrations, most of the OSA integration modules are stored in a DLL file (Windows) or shared library (UNIX). This setting tells MULTI the integration module's name and location. If no file extension is specified in *osa_integration_module_name*, MULTI appends **.dll** (Windows) or **.so** (UNIX) to it automatically.

If the option is absent and the OSA integration is not built into MULTI, the default name (the lowercase name for the OSA integration) will be used.

You can set the OSA integration module's name and location with this option. If the integration module is specified with only a base name, MULTI first searches for it in the MULTI installation directory, and then in a way defined by the host machine.

You can override the setting here by using the **-osa** MULTI command line option (see “Starting MULTI for Freeze-Mode Debugging and OS-Awareness” on page 451). Alternatively, use the **osasetup** command (see “Object Structure Awareness (OSA) Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book).

The Memory Access Block Size

Format:

OSA_CONFIG:OSA_MEMORY_ACCESS_BLOCK_SIZE:*number_in_bytes*

This option allows you to customize the memory access blocks size between MULTI and the target to improve the performance of **OSA Explorer**.

The default memory access block size of MULTI is 64 bytes.

Log Interaction

Format:

OSA_CONFIG:OSA_DEBUGGING_LOGFILE:*log_file*

This option allows you to see the interaction between MULTI and the OSA integration module. All interactions will be saved in the file *log_file*.

In addition to the communication between MULTI and the OSA integration module, MULTI will also print some diagnostic information for the OSA integration module into the log file.

The Object Type Definition

Format:

```
OSA_OBJECT_TYPE:object_name:unique_identifier  
:OSA_GLOBAL|OSA_NOT_GLOBAL
```

Before you specify the detailed information about an object, it must be assigned a unique identifier.

The argument, *object_name*, is the string to be shown in the **OSA Explorer**. It can be OSA_TASK (for the *task* object) or a string for another object. You can change an object's name into any string you like.

An object can be either a global object (OSA_GLOBAL) or a local object (OSA_NOT_GLOBAL). A global object does not require any special context for the OSA integration module to access it, but a local object requires a global object as context in order to access it. So, only global objects are displayed as tabs in the **OSA Explorer**; local objects can be used only as references to other objects.

The Object Definition

A *task* is a special type of object. If there is no definition for it in the configuration file, MULTI will not be able to provide the multitasking debugging feature, but OS-awareness is still available for the objects defined in the configuration file.

An object's information contains three parts: references, attributes, and additional shortcut menu entries.

The Reference Specification

Format:

```
master_object_name:OSA_REFERENCE_LIST:reference_object_name  
:reference_id:reference_name
```

An object (the master object) can have more than one reference object, and each reference object is defined by a statement using this syntax.

The string *reference_id* should be unique for all references of the *master_object_name*.

The string *reference_name* will be displayed in the drop-down list above the reference pane in the **OSA Explorer**.

The Object Attribute Definition

Each object has a set of attributes. Some attributes are universal and important to objects for various OSA integrations. For example, tasks have the *identifier*, *name*, *status*, and *executable* attributes, while other objects just have the *identifier* attribute.

An object must have an *identifier* attribute. If one is not defined for an object, the configuration file is invalid.

Each attribute of an object is defined with a statement in the following syntax.

Format:

```
object_name:OSA_ATTRIBUTE_COLUMN:  
[OSA_ID=|OSA_NAME=|OSA_STATUS=|OSA_EXEC=]attribute_name  
:attribute_id:OSA_SHOW|OSA_NO_SHOW:sorting_type
```

OSA_ID=, OSA_NAME=, OSA_STATUS=, and OSA_EXEC= are used to tell MULTI that the defined attributes are the *identifier*, *name*, *status*, or *executable*, respectively, of the corresponding object. When Task Debugger mode is enabled, task *identifier* and task *name* are shown in the title of the Debugger window, and task *status* is shown in the target list's **Status** column. If these attributes are not specified, MULTI uses default values in these places.

The string *attribute_id* must be unique for each attribute of an object.

OSA_SHOW and OSA_NO_SHOW tell MULTI which attribute columns are to be initially shown in the **OSA Explorer**.

sorting_type tells MULTI how to sort each column when it is clicked. The following are the valid sorting types:

- OSA_LONG — Sorts the corresponding column by long integer
- OSA_STRING — Sorts the corresponding column by string

- OSA_DATE — Sorts the corresponding column by date
- OSA_FILENAME — Sorts the corresponding column by filename
- OSA_FLOAT — Sorts the corresponding column by float value
- OSA_ADDRESS — Sorts the corresponding column by address (unsigned integers)

If no sorting type is defined for an attribute, its column will be sorted by string.

Shortcut Menu Entry Definitions

You can also define additional entries of the right-click shortcut menu in the object list pane of an **OSA Explorer**. Each such entry is defined by a statement in the following syntax.

Format:

object_name:OSA_MENU:*menu_entry_string*:*multi_commands*

Where *menu_entry_string* is the name to be displayed on the shortcut menu, and *multi_commands* is the command or commands to be executed whenever the menu option is selected.

If *menu_entry_string* is "`\x01`", it defines a menu separator. For example, you can add a menu separator into **Semaphore** list's shortcut menu with the following line:

```
Semaphore:OSA_MENU:"\x01":{ }
```

Example: Configuration File

The following is the configuration file used by INTEGRITY.

```
// OSA Integration Package Configuration File for INTEGRITY
// Copyright (C) 2000-Present Green Hills Software, Inc.

OSA_CONFIG:OSA_MULTI_MENU:INTEGRITY OSA Explorer...
OSA_CONFIG:OSA_EXPLORER_TITLE:INTEGRITY OSA Explorer (MULTI target ID 0x%x)
// Uncomment the following line to customize the memory access block
// size(in bytes) to turn up performance for your target.
// OSA_CONFIG:OSA_MEMORY_ACCESS_BLOCK_SIZE:512
```

```
// Uncomment the following line to generate an OSA log file
// OSA_CONFIG:OSA_DEBUGGING_LOGFILE:"integrity.log"

// Object Type List Section
// =====

// Define Object Type IDs and configure them for global display
// Format: OSA_OBJECT_TYPE:object name:object type id:global_display
OSA_OBJECT_TYPE:"Address Space":0:OSA_GLOBAL
OSA_OBJECT_TYPE:OSA_TASK:1:OSA_GLOBAL
OSA_OBJECT_TYPE:Connection:2:OSA_GLOBAL
OSA_OBJECT_TYPE:Activity:3:OSA_GLOBAL
OSA_OBJECT_TYPE:Semaphore:4:OSA_GLOBAL
OSA_OBJECT_TYPE:"Memory Region":5:OSA_GLOBAL
OSA_OBJECT_TYPE:Link:6:OSA_GLOBAL
OSA_OBJECT_TYPE:Clock:7:OSA_GLOBAL
OSA_OBJECT_TYPE:"I/O Device":8:OSA_GLOBAL
OSA_OBJECT_TYPE:Object:9:OSA_GLOBAL

// Address Space Window Format Sections
// =====
// Column Line Format:
//   type name:OSA_ATTRIBUTE_COLUMN:attribute name:attribute id:display:type
// List Line Format:
//   type name:OSA_REFERENCE_LIST:list type name:list id:list name
"Address Space":OSA_REFERENCE_LIST:OSA_TASK:0:Tasks
"Address Space":OSA_REFERENCE_LIST:Connection:1:Connection
"Address Space":OSA_REFERENCE_LIST:Activity:2:Activity
"Address Space":OSA_REFERENCE_LIST:Semaphore:3:Semaphore
"Address Space":OSA_REFERENCE_LIST:"Memory Region":4:"Memory Region"
"Address Space":OSA_REFERENCE_LIST:Link:5:Link
"Address Space":OSA_REFERENCE_LIST:Clock:6:Clock
"Address Space":OSA_REFERENCE_LIST:"I/O Device":7:"I/O Device"
"Address Space":OSA_REFERENCE_LIST:Object:8:Objects

"Address Space":OSA_ATTRIBUTE_COLUMN:OSA_ID="Domain ID":0:OSA_SHOW:OSA_ADDRESS
"Address Space":OSA_ATTRIBUTE_COLUMN:Name:1:OSA_SHOW:OSA_STRING
"Address Space":OSA_ATTRIBUTE_COLUMN:Virtual:2:OSA_SHOW:OSA_STRING
"Address Space":OSA_ATTRIBUTE_COLUMN:Objects:3:OSA_SHOW:OSA_LONG
"Address Space":OSA_MENU:"View AddressSpace Internals":if ($_OSA_OBJ_ID) \
    {substitute view (struct DomainStruct *)%EVAL{print /x $_OSA_OBJ_ID}; \
    } else {print "Domain ID is invalid.\n";}
```

```
// Task Space Window Format Section
// =====
// Column Line Format:
//   OSA_TASK:OSA_ATTRIBUTE_COLUMN:attribute name:attribute id:display:type
OSA_TASK:OSA_REFERENCE_LIST:Activity:0:"Interrupt Stack"
OSA_TASK:OSA_REFERENCE_LIST:Activity:1:"Other Activities"
OSA_TASK:OSA_REFERENCE_LIST:Semaphore:2:"Owned Binary Semaphores"
OSA_TASK:OSA_REFERENCE_LIST:Semaphore:3:"Owned HL Semaphores"
OSA_TASK:OSA_ATTRIBUTE_COLUMN:OSA_NAME=Name:0:OSA_SHOW:OSA_STRING
OSA_TASK:OSA_ATTRIBUTE_COLUMN:OSA_ID=TaskID:1:OSA_SHOW:OSA_ADDRESS
OSA_TASK:OSA_ATTRIBUTE_COLUMN:OSA_STATUS=Status:2:OSA_SHOW:OSA_STRING
OSA_TASK:OSA_ATTRIBUTE_COLUMN:Priority:3:OSA_SHOW:OSA_LONG
OSA_TASK:OSA_ATTRIBUTE_COLUMN:Weight:10:OSA_SHOW:OSA_LONG
OSA_TASK:OSA_ATTRIBUTE_COLUMN:Stack Start:4:OSA_NO_SHOW:OSA_ADDRESS
OSA_TASK:OSA_ATTRIBUTE_COLUMN:Stack End:5:OSA_NO_SHOW:OSA_ADDRESS
OSA_TASK:OSA_ATTRIBUTE_COLUMN:Stack HWM/Size:6:OSA_SHOW:OSA_STRING
OSA_TASK:OSA_ATTRIBUTE_COLUMN:"Address Space":7:OSA_SHOW:OSA_STRING
OSA_TASK:OSA_ATTRIBUTE_COLUMN:OSA_EXEC=Executable:8:OSA_NO_SHOW:OSA_STRING
OSA_TASK:OSA_ATTRIBUTE_COLUMN:Object Index:9:OSA_SHOW:OSA_STRING
//OSA_TASK:OSA_ATTRIBUTE_COLUMN:Stack HWM:10:OSA_SHOW:OSA_STRING
OSA_TASK:OSA_MENU:"View Task Internals":if ($_OSA_OBJ_ID) \
    {substitute view (struct TaskContextStruct *)%EVAL{print /x $_OSA_OBJ_ID}; \
    } else {print "Task ID is invalid.\n";}
OSA_TASK:OSA_MENU:"\x01":{}
OSA_TASK:OSA_MENU:Debug Task:if ($_OSA_OBJ_ID) \
    {osatask $_OSA_OBJ_ID} \
    else {print "Task ID is invalid.\n";}

//Clocks Window Format Section
Clock:OSA_ATTRIBUTE_COLUMN:OSA_ID="Clock ID":0:OSA_SHOW:OSA_ADDRESS
Clock:OSA_ATTRIBUTE_COLUMN:Name:1:OSA_SHOW:OSA_STRING
Clock:OSA_ATTRIBUTE_COLUMN:"Address Space":2:OSA_SHOW:OSA_STRING
Clock:OSA_ATTRIBUTE_COLUMN:Object Index:3:OSA_SHOW:OSA_STRING
Clock:OSA_MENU:"View Clock Internals":if ($_OSA_OBJ_ID) \
    {substitute view (struct ClockStruct *)%EVAL{print /x $_OSA_OBJ_ID}; \
    } else {print "Clock ID is invalid.\n";}

//IODevice Window Format Section
"I/O Device":OSA_ATTRIBUTE_COLUMN:OSA_ID="Device ID":0:OSA_SHOW:OSA_ADDRESS
"I/O Device":OSA_ATTRIBUTE_COLUMN:Name:1:OSA_SHOW:OSA_STRING
"I/O Device":OSA_ATTRIBUTE_COLUMN:Object Index:2:OSA_SHOW:OSA_STRING
"I/O Device":OSA_MENU:"View I/O Device Internals":if ($_OSA_OBJ_ID) \
    {substitute view (struct IIODeviceStruct *)%EVAL{print /x $_OSA_OBJ_ID}; \
    } else {print "I/O Device ID is invalid.\n";}

//Generic Object List format section
Object:OSA_REFERENCE_LIST:Link:0:"Links to Object"
Object:OSA_ATTRIBUTE_COLUMN:OSA_ID="Object ID":0:OSA_SHOW:OSA_STRING
Object:OSA_ATTRIBUTE_COLUMN:Type:1:OSA_SHOW:OSA_STRING
Object:OSA_ATTRIBUTE_COLUMN:"Address Space":2:OSA_SHOW:OSA_STRING
Object:OSA_MENU:"View Object Internals":if ($_OSA_OBJ_ID) \
    {substitute view (struct ObjectStruct *)%EVAL{print /x $_OSA_OBJ_ID}; \
    } else {print "Object ID is invalid.\n";}
```

```
//Activity list format section
Activity:OSA_ATTRIBUTE_COLUMN:OSA_ID="Activity ID":0:OSA_SHOW:OSA_ADDRESS
Activity:OSA_ATTRIBUTE_COLUMN:"Object Index":1:OSA_SHOW:OSA_STRING
Activity:OSA_ATTRIBUTE_COLUMN:Status:2:OSA_SHOW:OSA_STRING
Activity:OSA_ATTRIBUTE_COLUMN:Priority:3:OSA_SHOW:OSA_LONG
Activity:OSA_ATTRIBUTE_COLUMN:"Address Space":4:OSA_SHOW:OSA_STRING
Activity:OSA_ATTRIBUTE_COLUMN:Task:5:OSA_SHOW:OSA_STRING
Activity:OSA_ATTRIBUTE_COLUMN:"Waiting On":6:OSA_SHOW:OSA_STRING
Activity:OSA_MENU:"View Activity Internals":if ($_OSA_OBJ_ID) \
    {substitute view (struct ActivityStruct *)%EVAL{print /x $_OSA_OBJ_ID}; \
    } else {print "Activity ID is invalid.\n";}

//Semaphore window format section
Semaphore:OSA_ATTRIBUTE_COLUMN:OSA_ID="Semaphore ID":0:OSA_SHOW:OSA_ADDRESS
Semaphore:OSA_ATTRIBUTE_COLUMN:"Object Index":1:OSA_SHOW:OSA_STRING
Semaphore:OSA_ATTRIBUTE_COLUMN:Type:3:OSA_SHOW:OSA_STRING
Semaphore:OSA_ATTRIBUTE_COLUMN:Owner:2:OSA_SHOW:OSA_STRING
Semaphore:OSA_ATTRIBUTE_COLUMN:Value:5:OSA_SHOW:OSA_STRING
Semaphore:OSA_ATTRIBUTE_COLUMN:Priority:6:OSA_SHOW:OSA_STRING
Semaphore:OSA_ATTRIBUTE_COLUMN:"Address Space":4:OSA_SHOW:OSA_STRING
Semaphore:OSA_MENU:"View Semaphore Internals":if ($_OSA_OBJ_ID) \
    {substitute view (struct SemaphoreStruct *)%EVAL{print /x $_OSA_OBJ_ID}; \
    } else {print "Semaphore ID is invalid.\n";}
Semaphore:OSA_MENU:"\x01":{}
Semaphore:OSA_MENU:"Take Semaphore":if ($_OSA_OBJ_ID) \
    {substitute dialogue TakeSemaphore %EVAL{print /x $_OSA_OBJ_ID}; \
    } else {print "Semaphore ID is invalid.\n";}
Semaphore:OSA_MENU:"Release Semaphore":if ($_OSA_OBJ_ID) \
    {substitute osainject -ObjType "Semaphore" -ObjId %EVAL{print /x $_OSA_OBJ_ID}
    -MsgType "Release"; \
    } else {print "Semaphore ID is invalid.\n";}

//Link list format section
Link:OSA_ATTRIBUTE_COLUMN:OSA_ID="Link ID":0:OSA_SHOW:OSA_ADDRESS
Link:OSA_ATTRIBUTE_COLUMN:"Address Space":1:OSA_SHOW:OSA_STRING
Link:OSA_ATTRIBUTE_COLUMN:"Target Type":2:OSA_SHOW:OSA_STRING
Link:OSA_ATTRIBUTE_COLUMN:"Target ID":3:OSA_SHOW:OSA_STRING
Link:OSA_ATTRIBUTE_COLUMN:"Object Index":4:OSA_SHOW:OSA_STRING
Link:OSA_ATTRIBUTE_COLUMN:"Target Index":5:OSA_SHOW:OSA_STRING
Link:OSA_MENU:"View Link Internals":if ($_OSA_OBJ_ID) \
    {substitute view (struct LinkStruct *)%EVAL{print /x $_OSA_OBJ_ID}; \
    } else {print "Link ID is invalid.\n";}

//MemoryRegion list format section
"Memory Region":OSA_ATTRIBUTE_COLUMN:OSA_ID="Memory Region ID":0:OSA_SHOW:OSA_ADDRESS
"Memory Region":OSA_ATTRIBUTE_COLUMN:"Address Space":1:OSA_SHOW:OSA_STRING
"Memory Region":OSA_ATTRIBUTE_COLUMN:Beginning:2:OSA_SHOW:OSA_ADDRESS
"Memory Region":OSA_ATTRIBUTE_COLUMN:End:3:OSA_SHOW:OSA_ADDRESS
"Memory Region":OSA_ATTRIBUTE_COLUMN:"Object Index":4:OSA_SHOW:OSA_STRING
"Memory Region":OSA_MENU:"View Memory Region Internals":if ($_OSA_OBJ_ID) \
    {substitute view (struct MemoryRegionStruct *)%EVAL{print /x $_OSA_OBJ_ID}; \
    } else {print "Memory Region ID is invalid.\n";}
```



```
//Connection list format section
Connection:OSA_ATTRIBUTE_COLUMN:OSA_ID="Connection ID":0:OSA_SHOW:OSA_ADDRESS
Connection:OSA_ATTRIBUTE_COLUMN:"Address Space":1:OSA_SHOW:OSA_STRING
Connection:OSA_ATTRIBUTE_COLUMN:"Other End":2:OSA_SHOW:OSA_STRING
Connection:OSA_ATTRIBUTE_COLUMN:"AS of Other End":3:OSA_SHOW:OSA_STRING
Connection:OSA_ATTRIBUTE_COLUMN:"Object Index":4:OSA_SHOW:OSA_STRING
Connection:OSA_ATTRIBUTE_COLUMN:"ObjIndex of Other End":5:OSA_SHOW:OSA_STRING
Connection:OSA_MENU:"View Connection Internals":if ($_OSA_OBJ_ID) \
    {substitute view (struct ConnectionStruct *)%EVAL{print /x $_OSA_OBJ_ID}; \
    } else {print "Connection ID is invalid.\n";}
Connection:OSA_MENU:"\x01":{}
Connection:OSA_MENU:"Inject Message":if ($_OSA_OBJ_ID) \
    {substitute dialogue GetMessageToConnection %EVAL{print /x $_OSA_OBJ_ID}; \
    } else {print "Connection ID is invalid.\n";}
```


Establishing Serial Connections

Chapter 20

This Chapter Contains:

- Starting the Serial Terminal Emulator (MTerminal)
- Using Quick Connect
- Creating and Configuring Serial Connections
- The MTerminal Window
- Running the Serial Terminal Emulator in Console Mode
- mterminal Syntax and Arguments

There are many cases in which it may be necessary or convenient to establish a serial connection. You can do this easily by using MULTI's serial terminal emulator (**MTerminal**), which you can access from many of the MULTI tools or run as a stand-alone program. You can connect to a local serial port with **MTerminal's** Quick Connect, or you can configure and save multiple Serial Connection Methods, which you can later use to establish serial connections. This chapter describes how to use the serial terminal emulator and how to create, save, and use Serial Connection Methods.

Starting the Serial Terminal Emulator (MTerminal)

The following list describes different ways you can access the **MTerminal** serial terminal emulator. (The list makes frequent mention of the **Serial Connection Chooser**, the **MTerminal** window, and the Debugger's **Srl** pane. For more information about these interfaces, see "Using the Serial Connection Chooser" on page 484, "The MTerminal Window" on page 488, and "The Serial Terminal Pane" on page 49.)

- From the MULTI Launcher, click  and then select **Open Terminal** from the drop-down menu that appears. Alternatively, select **Components** → **Open Serial Terminal** from the menu bar. These actions start the **Serial Connection Chooser**, which allows you to create or edit a Serial Connection Method and then connect to the serial port.
- From the MULTI Launcher, click  and then select the name of a previously created Serial Connection Method from the drop-down menu that appears. This starts the **MTerminal** window with a terminal connected to the specified serial port. If the connection is unsuccessful, an error message appears.
- From the command line, execute **mterminal** (for command usage, see "mterminal Syntax and Arguments" on page 492). This opens the serial terminal emulator as a stand-alone program. The emulator usually opens in graphical mode, displaying either the **MTerminal** window or the **Serial Connection Chooser**, depending on the arguments specified. On UNIX hosts, you can also open the emulator in console mode. For more information, see "Running the Serial Terminal Emulator in Console Mode" on page 492.
- From the MULTI Debugger, select **Tools** → **Serial Terminal** → **Make Serial Connection**. Alternatively, issue the **serialconnect** command with no arguments in the command pane. (For information about the **serialconnect**

command, see “Serial Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.) These actions start the **Serial Connection Chooser**, which allows you to create or edit a Serial Connection Method and then connect to a serial port. If the connection is successful, the **Srl** tab becomes available in the bottom pane of the Debugger. Selecting the **Srl** tab displays a terminal connected to the specified serial port. If the serial connection is unsuccessful, an error message appears and the **Srl** tab does not become available.

- From the MULTI Debugger, select **Tools** → **Serial Terminal** and then select the name of a previously created Serial Connection Method from the drop-down list that appears. Alternatively, issue the **serialconnect** command with arguments specifying the parameters for the connection in the command pane. (For information about the **serialconnect** command, see “Serial Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.) If the connection is successful, the **Srl** tab becomes available in the bottom pane of the Debugger. Selecting the **Srl** tab displays a terminal connected to the specified serial port. If the serial connection is unsuccessful, an error message appears and the **Srl** tab does not become available.

Using Quick Connect

You can connect to a local serial port easily and quickly by using **MTerminal's** Quick Connect. With Quick Connect, you do not need to create and configure a connection. To use Quick Connect, open an **MTerminal** window and perform the following steps:

1. Select an available baud rate from the **Baud Rate** drop-down list.
2. Select one of the commonly used ports from the **Port** drop-down list, or type the name of a port in the **Port** field.
3. Click the **Quick Connect** button.

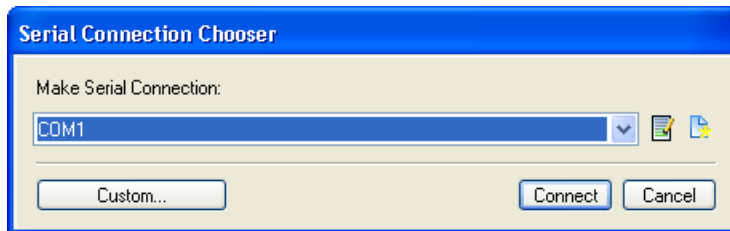
For information about serial connections that you can configure and save, see the next section.

Creating and Configuring Serial Connections

MULTI provides two graphical tools, the **Serial Connection Chooser** and the **Serial Connection Settings** dialog box, that allow you to configure and save one or more Serial Connection Methods and use those methods to establish serial connections and open terminal emulator windows. (These tools are analogous to the **Connection Chooser** and **Connection Editor**, which allow you to configure and save Connection Methods for connecting to your target. See Chapter 4, “Connecting to Your Target”.) The **Serial Connection Chooser** and the **Serial Connection Settings** dialog are described in the following sections.

Using the Serial Connection Chooser

You can use the **Serial Connection Chooser** to create a new Serial Connection Method, edit an existing Serial Connection Method, or establish a serial connection using a saved Serial Connection Method. For information about opening the **Serial Connection Chooser**, see “Starting the Serial Terminal Emulator (MTerminal)” on page 482.



To create a new standard Serial Connection Method from the **Serial Connection Chooser**, click . To edit an existing Serial Connection Method, select the Method from the drop-down list and click . Both of these actions open a **Serial Connection Settings** dialog, which allows you to configure your new or existing Serial Connection Method. For more information, see “Using the Serial Connection Settings Dialog” on page 485.

After you have created at least one Serial Connection Method, you can establish a connection by selecting your desired method from the drop-down list and then clicking **Connect**.

You can also create a Custom Serial Connection Method by entering a command in the **Serial Connection Chooser**, rather than using the graphical **Serial Connection Settings** dialog. To do this, click the **Custom** button in the **Serial**

Connection Chooser and use the syntax given below to describe the connection in the **Make Serial Connection** text field of the Chooser.

serialconnect *port_name* [*mterminal_parameters*]

This command accepts the same arguments as the **mterminal** command, which is used to launch **MTerminal** as a stand-alone application. For a description of the arguments that can be used to set the parameters of your connection, see “mterminal Syntax and Arguments” on page 492.

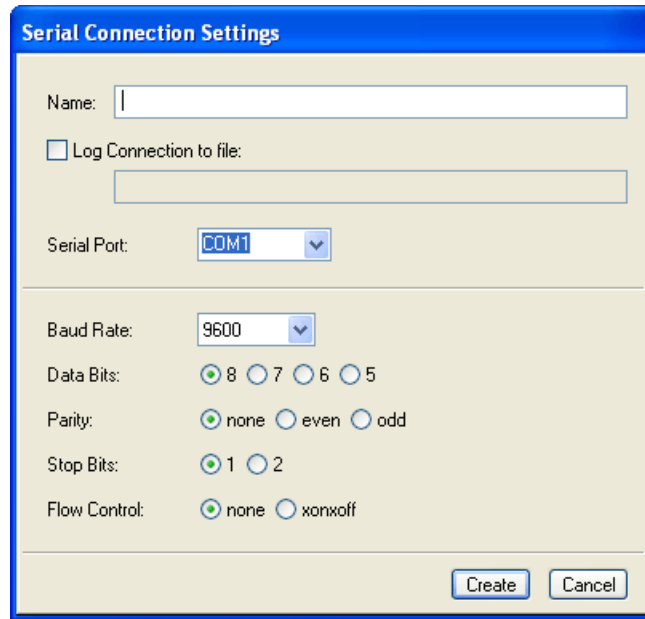
Saved serial connections are stored in *user_dir***Application Data\GHS\serialconnection.odb** on Windows and in *user_dir*/.ghs/**serialconnection.odb** on UNIX.

Using the Serial Connection Settings Dialog

Like the Connection Methods described in Chapter 4, “Connecting to Your Target”, Serial Connection Methods serve as templates that specify the settings MULTI should use to establish a particular serial connection. You can use the **Serial Connection Settings** dialog to configure a Serial Connection Method.

To create a new Serial Connection Method or change the configuration of an existing Serial Connection Method:

1. Open the **Serial Connection Chooser** (see “Starting the Serial Terminal Emulator (MTerminal)” on page 482).
2. Click  to create a new connection or  to edit an existing connection. This will open a **Serial Connection Settings** dialog.



3. Enter or change the name of your Serial Connection Method. If no name is entered, the method you create will be a Temporary Serial Connection Method and will exist only for the duration of the current **MTerminal** session.
4. Enter or change the configuration settings for your Serial Connection Method. For a description of the setting options, see “Serial Connection Settings” on page 487.
5. Click **Create** or **Apply** to save the Serial Connection Method (the button label displayed will depend on whether you are creating a new Method or editing an existing one).
6. Close the **Serial Connection Settings** dialog. The new (or modified) connection will now be saved and displayed in the **Serial Connection Chooser** drop-down list.

Saved serial connections are stored in *user_dir***Application Data**\GHS\serialconnection.odb on Windows and in *user_dir*/.ghs/serialconnection.odb on UNIX.

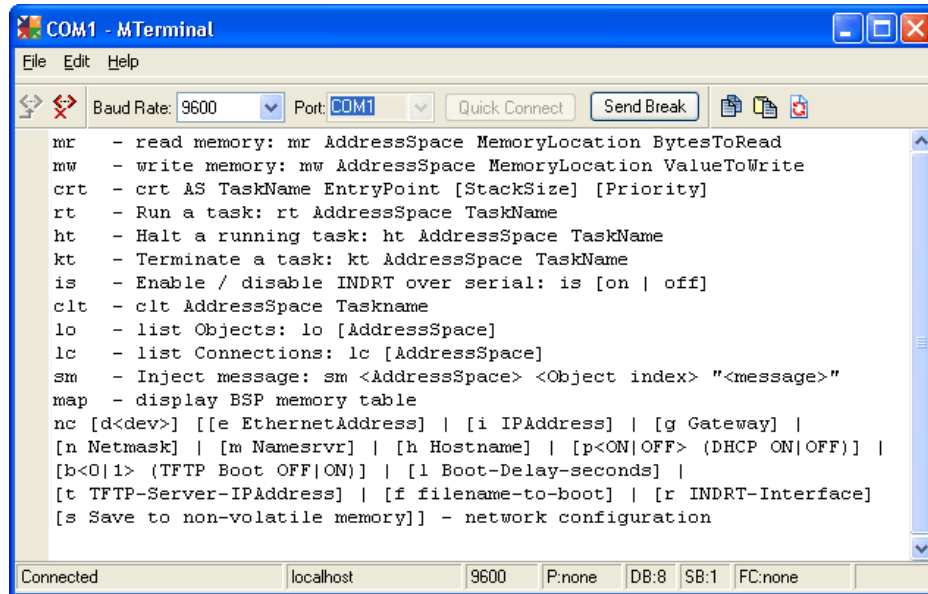
Serial Connection Settings

The following table describes the various settings that can be edited from the **Serial Connection Settings** dialog.

Name Specifies the name of the connection. If no name is entered, the method you create will be a Temporary Serial Connection Method and will exist only for the duration of the current MTerminal session.
Log Connection to file Logs the output from the serial port to the specified file. By default, this option is cleared (i.e., logging is disabled).
Serial Port Identifies the name of the serial port to connect to. The default values in the list are determined by the local host. If none of the values are applicable, you can type in the appropriate name. (Serial port names can be prepended with /dev.)
Baud Rate Specifies the baud rate for your connection. This setting defaults to 9600 .
Data Bits Specifies the data bits for your connection. This setting defaults to 8 .
Parity Specifies the parity for your connection. This setting defaults to none .
Stop Bits Specifies the stop bits for your connection. This setting defaults to 1 .
Flow Control Specifies the flow control for your connection. This setting defaults to none .

The MTerminal Window

The **MTerminal** window is a serial terminal window that opens when you use one of the methods described in “Starting the Serial Terminal Emulator (MTerminal)” on page 482.



You can use the main terminal window to interact with a serial port to which you have connected. This window provides full **VT100** support.

The MTerminal Menu Bar

The **MTerminal** menu bar contains three menus: the **File** menu, **Edit** menu, and **Help** menu. The following sections describe the items available from each of these menus.

The File Menu

The table below lists the items available in the **File** menu.

Item	Meaning
Connect	Opens the Serial Connection Chooser , which allows you to create and/or edit Serial Connection Methods. For more information, see “Using the Serial Connection Chooser” on page 484. This menu item is only available if the MTerminal window is not connected to any serial port.
Disconnect	Closes the current serial connection. This menu item is only available if the MTerminal window is connected to a serial port.
Close MTerminal	Closes the MTerminal window.

The Edit Menu

The table below lists the items available in the **Edit** menu.

Item	Meaning
Baud Rate	Opens a submenu that allows you to adjust the baud rate of a serial connection dynamically. The submenu items are only available if a serial connection has been established.
Send Break	Sends a serial break to the serial port. This menu item is only available if a serial connection has been established.
Copy	Copies text selected in the terminal window.
Paste	Pastes previously copied text into the terminal window. This menu item is only available if a serial connection has been established.
Clear	Clears text from the terminal window.

The Help Menu

The table below lists the items available in the **Help** menu.

Item	Meaning
MTerminal Help	Displays online help information about MTerminal .
About	Shows basic information about MTerminal , including the version number and revision date.

The MTerminal Toolbar

The **MTerminal** toolbar contains buttons and drop-down lists that allow you to connect to a serial port, disconnect from a serial port, Quick Connect, and copy, paste, and clear text in the terminal window. The following table describes the buttons and fields available on the **MTerminal** toolbar.

Item	Meaning
 Connect	Establishes a serial connection. This button is only available if the MTerminal window is not connected to any serial port. Clicking this button opens a drop-down menu that contains the following selections: • Open Terminal — Opens the Serial Connection Chooser, which allows you to create and/or edit Serial Connection Methods. For more information, see “Using the Serial Connection Chooser” on page 484. • <i>Recently used Serial Connection Methods</i> — Attempts to establish a connection to the serial port described in the selected method.
 Disconnect	Closes the current serial connection. This button is only available if the MTerminal window is connected to a serial port.
Baud Rate	Allows you to adjust the baud rate of a serial connection dynamically.
Port	Specifies the local serial port to use for Quick Connect.
Quick Connect	Connects to a local serial port based on the information selected in the Baud Rate and Port drop-down lists.

Item	Meaning
Send Break	Sends a serial break to the serial port. This button is only available if a serial connection has been established.
 Copy	Copies text selected in the terminal window.
 Paste	Pastes previously copied text into the terminal window. This button is only available if a serial connection has been established.
 Clear	Clears text from the terminal window.

The MTerminal Status Bar

The **MTerminal** status bar lists information about the parameters of the serial connection. Status bar information, as it is displayed from left to right, follows:

- The status of the connection (either **Connected** or **Disconnected**)
- The name of the host on which the serial port is located (or **localhost** if the port is on the local host)
- The baud rate
- The parity (prepended by **P:**)
- The data bits (prepended by **DB:**)
- The stop bits (prepended by **SB:**)
- The flow control (prepended by **FC:**)
- The name of the serial port

Running the Serial Terminal Emulator in Console Mode

MTerminal can be run in a non-graphical console mode on UNIX hosts. This might be useful when running validations. To run **MTerminal** in console mode, launch the serial terminal emulator from the command line using the **-nodisplay** option. For example, to connect to the `ttyS0` port using the default parameters for the serial settings, you would enter the command:

```
mterminal ttyS0 -nodisplay
```



Note Console mode does not have **VT100** support.

mterminal Syntax and Arguments

To launch **MTerminal** as a stand-alone application, run the **mterminal** command from the command line. The command usage is:

```
mterminal port_name [mterminal_parameters]
```

where *port_name* specifies what serial port is being used (for example, `ttya`, `ttyS0`, or `COM1`), and *mterminal_parameters* can be one or more of the options listed in the following table. (Parameters that are not specified use default values.)



Note The following arguments can also be used with the **serialconnect** command, which you can enter in the **Make Serial Connection** text field of the **Serial Connection Chooser**. For more information, see “Using the Serial Connection Chooser” on page 484.

-baud <i>baudrate</i>	Specifies the baud rate, where <i>baudrate</i> can be any one of the following: 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200, or 230400. The default is 9600.
-databits <i>DB</i>	Specifies the data bits, where <i>DB</i> can be 5, 6, 7, or 8. The default is 8.
-flowcontrol <i>FC</i>	Specifies flow control, where <i>FC</i> can be <code>none</code> or <code>xonxoff</code> . The default is <code>none</code> .
-help	Prints help information. This option is only valid in console mode and does not work on Windows hosts.

-log_file <i>filename</i>	Enables logging and specifies the name of the file to which data is written. By default, logging is disabled.
-nodisplay	Runs mterminal in console mode. This option causes all output to be printed to standard output and all input to be received from standard input. In console mode, no graphical windows appear and the serial port must be specified. This option does not work on Windows hosts.
-parity <i>P</i>	Specifies parity, where <i>P</i> can be <code>none</code> , <code>even</code> , or <code>odd</code> . The default is <code>none</code> .
-stopbits <i>SB</i>	Specifies stop bits, where <i>SB</i> can be either <code>1</code> or <code>2</code> . The default is <code>1</code> .

The following example starts the GUI version of **MTerminal** on the serial port `ttys0`. The baud rate is 38400.

```
mterminal ttys0 -baud 38400
```


Appendices

Part IV

Debugger GUI Reference

Appendix A

This Appendix Contains:

- Debugger Window Menus
- The Debugger Window Toolbar
- The Target List Shortcut Menu
- Source Pane Shortcut Menus
- The Command Pane Shortcut Menu
- The Source Pane Search Dialog Box
- The File Chooser Dialog Box (UNIX)

This appendix contains detailed descriptions of the menus and toolbar buttons available in the main Debugger window, as well as descriptions of some of the dialog boxes you can open via these menus and buttons. Most of these items are mentioned in the main text of this book in the context in which they are used. They are listed here together to provide a comprehensive reference.

For information about other GUI elements of the Debugger window, including descriptions of the target list, source pane, navigation bar, and information panes, see Chapter 3, “The Main Debugger Window”.



Note Some menu items, shortcuts, commands, and dialog box buttons open a file chooser that allows you to browse to and specify a file for a particular operation. For information about how to use a Windows file chooser, see your Windows documentation. For information about the UNIX file chooser, see “The File Chooser Dialog Box (UNIX)” on page 554.

Debugger Window Menus

The sections below describe the menu items that are accessible from the Debugger window’s menu bar.

Context-sensitive menu items are dimmed if they are unavailable in your current environment. For example, if you are debugging a running process, the **Go on Selected Items** item in the **Debug** menu is dimmed.

The menus on the main Debugger window, from left to right, are:

- The **File** menu (see page 499)
- The **Debug** menu (see page 503)
- The **View** menu (see page 512)
- The **Browse** menu (see page 518)
- The **Target** menu (see page 519)
- The **Tools** menu (see page 523)
- The **Config** menu (see page 525)
- The **Windows** menu (see page 530)
- The **Help** menu (see page 530)

The File Menu

The following table describes the selections available from the **File** menu in the main Debugger window.

Menu Item	Meaning
Debug Program as New Entry	Opens a file chooser from which you can select a program to debug. MULTI adds the program to the target list and loads it in the Debugger window. The resource file (if any) of the new program being debugged is executed and the final debugging environment is shared by the active Debugger window and secondary Debugger windows (if any). Corresponds to: the dbnew command (see Chapter 3, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Debug Program	Similar to Debug Program as New Entry, except that the previously debugged program is removed from the target list and terminated or detached. The resource file (if any) of the new program being debugged is executed to establish the final debugging environment. Any configuration changes made during the debugging session of the previous program remain in effect unless overridden. Corresponds to: the debug command (see Chapter 3, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Print	Opens the Print (Windows) or Print Setup (UNIX) dialog box. The dialog box allows you to print the current source file, including interlaced assembly code if it is the current display mode, but not including non-ASCII characters. If no high-level source file is available (that is, there is only assembly code), this option is unavailable. For details, see “The Print Setup Dialog Box” on page 501.
Print Window	Opens the Print (Windows) or Print Setup (UNIX) dialog box to print the visible, ASCII contents of the source pane. For details, see “The Print Setup Dialog Box” on page 501.
Write to File	Opens a file chooser for you to specify a file to save to. You can save the source file, including interlaced assembly code if it is the current display mode, but you cannot save non-ASCII characters. If no high-level source file is available (that is, there is only assembly code), only the contents of the source pane are saved. Click Save in the file chooser to perform the save.

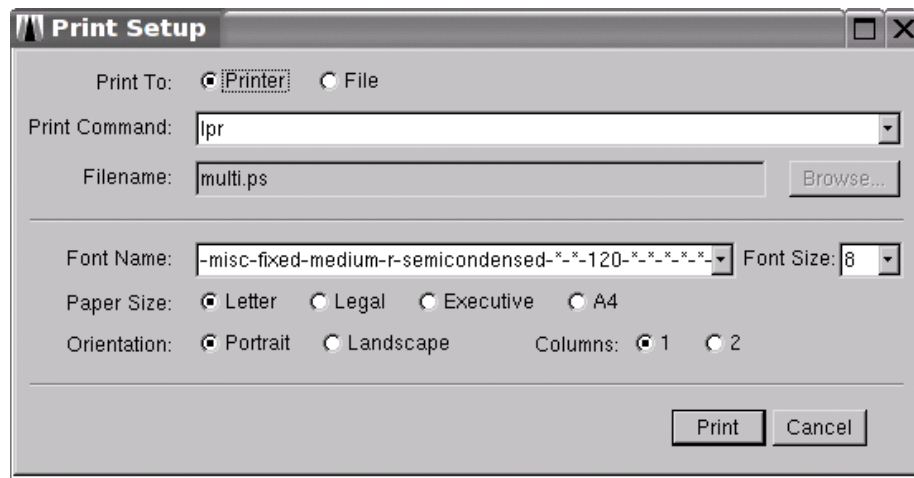
Menu Item	Meaning
Attach to Process	Opens a dialog box for you to specify the ID of a running task. This menu option only works with a multitasking target. Clicking OK in the dialog box adds a new target list entry for the selected process and loads the task in the Debugger window. Corresponds to: the attach command (see Chapter 3, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Detach from Process	In run mode, detaches the Debugger from the current process, and selects the next program in the target list. If you are not debugging in run mode, the entry for the current process is removed from the target list, but the debug server remains connected. All breakpoints are removed before detaching. Corresponds to: the detach command (see Chapter 3, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
1 pathname 2 pathname 3 pathname 4 pathname	Lists the most recent programs opened in the Debugger. To debug one of these in the current Debugger window, select it.
Close Entry	Removes the currently selected entry from the target list. If the entry is a connection, selecting this option disconnects from it. In run mode, MULTI detaches from a selected process. If you are not debugging in run mode, MULTI terminates a selected process. If the entry is the only remaining entry in the target list, all Debugger windows close and the Debugger exits. Corresponds to: Ctrl + W and the quit entry command (see Chapter 3, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Menu Item	Meaning
Close Debugger Window	Closes the current Debugger window, but does not detach from or terminate any process unless the current Debugger window is the only one remaining. If the current Debugger window is the only one remaining, MULTI detaches from run-mode processes and terminates non-run-mode processes when it closes the window. Corresponds to: , Ctrl + Q, and the quit window command (see Chapter 3, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Exit All	Closes all MULTI IDE windows and exits. Corresponds to: the quitall command (see Chapter 3, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

The Print Setup Dialog Box

On Windows, selecting a printing operation from the **File** menu opens a standard **Print** dialog box with settings and options appropriate for your version of Windows and your printer and printer driver. See your Windows and printer documentation for details about the settings on this dialog.

On UNIX systems, selecting a printing operation from the **File** menu opens the following **Print Setup** dialog box.



The settings of the **Print Setup** dialog box are described in the following table.

Item	Meaning
Print To	Specifies where to send the print output. Select either Printer (the default) or File (to write to a postscript file).
Print Command	Specifies the print command that will be run when you click Print. Enter the appropriate command and options for printing a file on your specific system (for example, lpr on UNIX). You can also set the print command with the configure printCommand command. For information about the configure command, see “Using the configure Command” in Chapter 8, “Configuring and Customizing MULTI” in the <i>MULTI: Editing Files and Configuring the IDE</i> book. This field is only available if you have chosen the Printer radio button above.
Filename	Specifies the output post-script file for print to file operations. This field is only available if you have selected the File radio button above. Enter the file to write to, or click Browse to open a chooser and navigate to an existing file. (See “The File Chooser Dialog Box (UNIX)” on page 554 for a description of the chooser.)
Font Name	Specifies the font for your printing operations. Select your desired font from the drop-down list, which contains a list of available fonts on your system. You can also type a font name into this field if it is not in the list.
Font Size	Specifies the font size used for printing. Select your desired size from the drop-down list. The default font size is 8 point.
Paper Size	Specifies your paper size. Select Letter , Legal , Executive , or A4 . The default setting is Letter .
Orientation	Specifies the layout for your printed document. Select Portrait or Landscape . The default setting is Portrait .
Columns	Specifies whether to print the output in one or two columns. The default setting is 1 .
Print	Executes the print request.
Cancel	Cancels the print request and discards any settings you have changed.

The Debug Menu

The following table describes the selections available from the **Debug** menu in the main Debugger window.

Menu Item	Meaning
Set Program Arguments	Opens a dialog box where you can specify the arguments for the current program. (Program arguments can only be passed to stand-alone applications that are started from MULTI.) The dialog box also allows you to control input and output redirection for the current process. See “The Arguments Dialog Box” on page 507. Corresponds to: the setargs command and the r command (see “General Program Execution Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Go on Selected Items	Starts or resumes items selected in the target list. This menu item cannot be used to start tasks on VxWorks systems. For information about starting tasks on these systems, see the runtask command in “Run Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. Corresponds to: , F5, and the c command (see “Continue Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Restart	Starts or restarts the current program being debugged, with preset arguments (if any). In TimeMachine mode, selecting this menu item restarts the process from the first traced instruction, but does not pass any arguments. If you select this menu item while debugging a Dynamic Download INTEGRITY application, MULTI attempts to (re)load the application. This menu item may not be available for relocatable modules. Corresponds to: , Ctrl + Shift + F5, and the restart command (see “Run Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

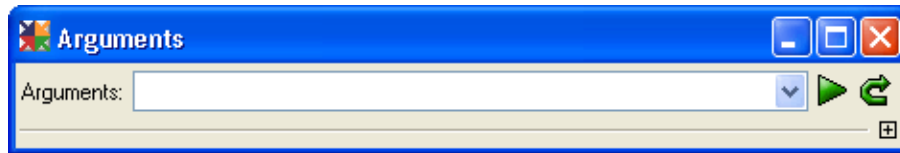
Menu Item	Meaning
Halt on Selected Items	Halts the items selected in the target list. Corresponds to:  and the halt command (see “Halt Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Kill Selected Items	Kills the items selected in the target list. Corresponds to: Shift + F5 and the k command (see “Halt Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Step (into Functions) on Selected Items	Executes single statements, stepping into procedure calls of items that are selected in the target list. Corresponds to: , F11, and the s command (see “Single-Stepping Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Next (over Functions) on Selected Items	Executes single statements, stepping over procedure calls of items that are selected in the target list. Corresponds to: , F10, and the n command (see “Single-Stepping Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Return on Selected Items	For items that are selected in the target list, continues execution to the end of the current subroutine and stops in the calling routine after returning to it. Corresponds to: , F9, and the cU command (see “Continue Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Run to Cursor	Continues execution to the location of the current line pointer. Corresponds to: Ctrl + F10
Halt System	Halts all tasks on the target system, thereby freezing the target. This menu item is not supported on all operating systems and only appears if a run-mode connection is selected in the target list. Corresponds to: the groupaction -h command (see Chapter 19, “Task Group Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Menu Item	Meaning
Run System	Restores the tasks on the target to the status they held before the system was halted. This menu item only appears if a run-mode connection is selected in the target list. Corresponds to: the groupaction -r command (see Chapter 19, “Task Group Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Send Signal	Opens a dialog box that specifies the signal name and then sends a signal to the current process. To list signal names, enter the l z (lowercase L lowercase Z) command. See the l command in Chapter 9, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
Set Watchpoint	Opens a dialog box that creates a watchpoint. Corresponds to: the watchpoint command (see Chapter 4, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Remove All Breakpoints	Deletes software breakpoints: • If the OSA master process is selected in the target list — Deletes all normal software breakpoints (except group breakpoints) in the master process. For more information, see “Working with Freeze-Mode Breakpoints” on page 462. • If a run-mode AddressSpace is selected in the target list and you are using INTEGRITY 10 or later — Deletes all any-task breakpoints in the AddressSpace. • If a task is selected in the target list — Deletes all task-specific software breakpoints in the task. Corresponds to: the D command (see Chapter 4, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Toggle All Breakpoints	Toggles the active state of all software breakpoints of the current program (i.e. all currently enabled breakpoints will be disabled, and all currently disabled breakpoints will be enabled). Corresponds to: the Tog command (see Chapter 4, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

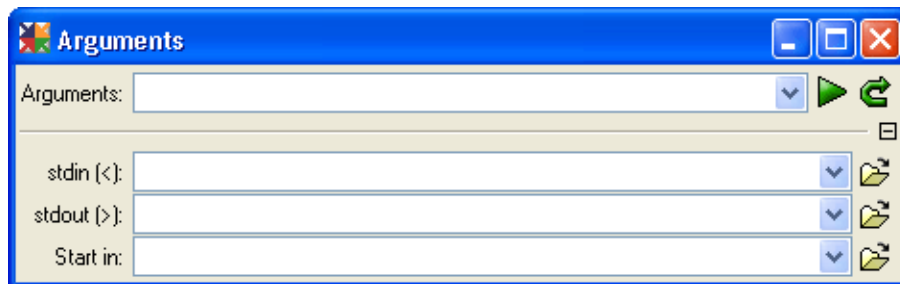
Menu Item	Meaning
Load Symbols	Loads symbols from a selected object, file, or library. Select the symbols to load by selecting Select Symbols to Load and locating the file manually, or by selecting one of the entries in the most recently loaded list at the bottom of the submenu. Corresponds to: the loadsym command (see Chapter 3, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Unload Symbols	Unloads the symbols for the selected object. Selecting this entry opens a submenu listing all of the currently loaded symbols. Selecting an entry in the menu unloads the corresponding set of symbols. If there are no sets or only one set of symbols loaded, the submenu indicates this (you may not unload the last set of symbols). Corresponds to: the unloadsym command (see Chapter 3, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Prepare Target	Opens a dialog box in which you can select a method for setting up your target. You can either download your program to RAM, flash your program to ROM, or verify that your program is already on the target. For more information, see “Preparing Your Target” on page 90. This menu item may not be available for relocatable modules. Corresponds to:  and the prepare_target command (see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Use Connection	Opens the Use Connection submenu. See “The Use Connection Submenu” on page 509.
Debug Settings	Opens the Debug Settings submenu. See “The Debug Settings Submenu” on page 510.
Target Settings	Opens the Target Settings submenu. See “The Target Settings Submenu” on page 511.

The Arguments Dialog Box

The **Arguments** dialog box opens when you select **Debug** → **Set Program Arguments**. When the dialog box first appears, it may be displayed in a contracted form:



To expand the dialog box to see the I/O setting fields, click the small plus sign icon (⊕) on the bottom right corner of the contracted dialog box. The expanded dialog box is shown below.



To hide the three I/O fields, click the minus sign (⊖) to the right of the line dividing the top and bottom sections of the dialog box.

The following table describes the items in the expanded **Arguments** dialog box.

Item	Meaning
Arguments	Specifies the arguments to be passed to the current program the next time you run it without specifying any arguments (for example, by clicking  or  , or by executing the r command, etc.). Program arguments can only be passed to stand-alone applications that are started from MULTI. If you specify any arguments in the next execution (with the r command, for example) the existing arguments in the Arguments field are replaced by the ones specified. These new arguments are displayed the next time the Arguments dialog box is opened.
The  button	Runs the program with the arguments specified in the Arguments text field.
The  button	Restarts the process with the arguments specified in the Arguments text field.

Item	Meaning
The  button	Closes the Arguments dialog box. Whether this button appears on your toolbar depends on the setting of the option Display close (x) buttons. To access this option, select Config → Options → General Tab.
stdin (<)	Specifies a file on the host system that will be used as input to your process. Enter a filename, or click  to open a file chooser to browse to an existing file. If this field is not specified, the input comes from standard input.
stdout (>)	Specifies a file on the host system that will capture output from your process. Enter a filename, or click  to open a file chooser to browse to an existing file. If this field is not specified, the output goes to standard output.
Start in	Specifies the directory in which the process runs or, for embedded processes that use host I/O, specifies the directory that MULTI uses to perform host I/O operations. Enter a directory, or click  to open a directory chooser to browse to an existing directory. For information about the command equivalent of this field, see the rundir command in “Run Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

The Use Connection Submenu

The following table describes the selections available from the **Debug → Use Connection** submenu.

Menu Item	Meaning
[Status] Connection_Name	Associates the current executable with the specified connection. Only compatible, currently active connections are listed. If a connection appears dimmed, the current executable cannot be associated with that connection, or it is already associated with that connection. Status is listed either as: Available (the connection can accept more executables), Current (the connection is associated with the current executable), or Full (using the connection will cause another executable to stop using it). For more information, see “Associating Your Executable with a Connection” on page 87.
Stop Using Current Connection	Disassociates the selected executable from the connection it is currently associated with. Selecting this menu item does not disconnect from the target debug server. For more information, see “Associating Your Executable with a Connection” on page 87. Corresponds to: the change_binding unbind command (see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Create New Connection	Opens the Connection Chooser , which you can use to establish a connection. For more information, see “Creating a Standard Connection Using the Connection Chooser” on page 58.

The Debug Settings Submenu

The following table describes the selections available from the **Debug** → **Settings** submenu.

Menu Item	Meaning
Attach on Fork/Thread	Toggles the option to attach to child processes spawned by calls to <code>fork()</code> . This option is only supported on native targets.
Memory Sensitive	Controls whether the Debugger attempts to limit memory reads from the target. See “Memory Sensitive Mode” on page 97.
Cache Memory Reads	Controls whether the Debugger uses a cache to read memory more efficiently. When enabled, the Debugger reads memory in blocks, which are stored in a cache. This reduces the number of memory accesses required. For example, with the cache active, printing a structure with ten members would require only one memory read as opposed to ten memory reads. However, this could cause problems when accessing volatile memory.
Disassemble From Host	Controls whether the Debugger displays disassembly for programs based on the code in the executable file, as opposed to the contents of memory. When enabled, the Debugger disassembles instructions found in the executable file. This has the advantage of not requiring any memory reads from the target. When disabled, the Debugger reads instructions from target memory when disassembling.
No Stack Trace	Controls whether the Debugger is allowed to access the stack for stack trace information. When the setting is enabled, stack traces are disallowed. This can be useful if you know that the stack does not contain a valid stack trace, and you want to prevent the Debugger from attempting to trace into invalid memory. See “No Stack Trace Mode” on page 98.
Auto Check Coherency	Controls whether the Debugger checks coherency automatically. When automatic coherency checking is enabled, MULTI checks the coherency of the specified number of addresses at every stop (see Number of Addresses to Check below). If it finds discrepancies between the contents of memory and what you loaded onto your target, it highlights the differing lines in the source pane. See “Checking Coherency Automatically” on page 418.

Menu Item	Meaning
Number of Addresses to Check	Opens a dialog box where you can enter the number of addresses you want checked for coherency on each stop. By default, MULTI checks either 16 addresses or the number of addresses lasting the length of 4 instructions (whichever is fewer). MULTI attempts to check an equal number of addresses before and after the current program counter; however, it does not cross procedure boundaries. For more information, see “Checking Coherency Automatically” on page 418. See also Auto Check Coherency above.
Fast Source Step	Controls whether the Debugger attempts to speed up stepping through source code. When enabled, on each source step, the Debugger analyzes the code, sets temporary breakpoints on all possible step destinations, and allows the process to run until one of the breakpoints is hit. This allows the process to run at nearly full speed during the source step. When disabled, the Debugger usually steps one machine instruction at a time until it reaches the next source line. However, even when disabled, the Debugger sets temporary breakpoints and runs to step over function calls. This method is generally much slower.

The Target Settings Submenu

The following table describes the selections available from the **Debug** → **Target Settings** submenu.

Menu Item	Meaning
Stop on Task Creation	Specifies whether you want the target’s operating system to halt each task as soon as it is created by the application.
Attach on Task Creation	Specifies whether you want to debug the newly created task. If enabled, each newly created task is halted and opened in the Debugger window. This option is applicable only when the Stop on Task Creation menu item is selected.

Menu Item	Meaning
Lock Other Threads on Single Step	Specifies whether you want the target's operating system to lock all threads other than the selected thread when performing a single step. When MULTI steps over a source line, it places temporary breakpoints on all possible exits from the source line and runs the target. If this option is enabled, which is the default, only the thread that you are single stepping will be running during the single step. If this option is disabled, other threads on the target may be running during the single step. This option is only available for Cafe OS.
Turn Off Target on Disconnect	Specifies whether you want MULTI to automatically turn off the target when you disconnect from the target. This option is only available for Cafe OS, and, if enabled, cafestop will be run when you disconnect from the target.

The View Menu

The following table describes the selections available from the **View** menu in the main Debugger window.

Menu Item	Meaning
Navigation	Opens the Navigation submenu. See “The Navigation Submenu” on page 515.
Display Mode	Opens the Display Mode submenu. See “The Display Mode Submenu” on page 516.
Breakpoints	Opens the Breakpoints window. See “The Breakpoints Window” on page 123. Corresponds to:  and the breakpoints command (see Chapter 4, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Call Stack	Opens a Call Stack window. See “Viewing Call Stacks” on page 376. Corresponds to:  and the callsview command (see Chapter 6, “Call Stack Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Menu Item	Meaning
Local Variables	Opens a Data Explorer displaying all local variables. See Chapter 9, “Viewing and Modifying Variables with the Data Explorer”. Corresponds to: , the view \$locals\$ command, and the localsview command (see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Registers	Opens a Register View window. See “The Register View Window” on page 234. Corresponds to:  and the regview command (see “Register Commands” in Chapter 7, “Configuration Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Memory	Opens an empty Memory View window. See Chapter 13, “Using the Memory View Window”. Corresponds to:  and the memview command (see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Memory Allocations	Opens the Memory Allocations window. See “Using the Memory Allocations Window” on page 325. Corresponds to: the heapview command (see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Profile	Opens the Profile window. See “The Profile Window” on page 348. Corresponds to: the profile command (see Chapter 13, “Profiling Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Menu Item	Meaning
Task Manager	Opens a Task Manager window if you are debugging in a run-mode environment. Otherwise it opens a Process Viewer. See “The Task Manager” on page 424 and “Viewing Native Processes” on page 378. If you are debugging in run mode, corresponds to: the taskwindow command (see “Object Structure Awareness (OSA) Commands” in Chapter 15, “Scripting Command Reference” in the <i>MULTI: Debugging Command Reference</i> book). If you are debugging in a native environment, corresponds to: the top command (see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
OSA Explorer	Opens an OSA Explorer on the current process in a freeze-mode debugging environment or on the current debug server in a run-mode debugging environment, or opens the Linux Threads window in Linux. The OSA Explorer shows information for objects recognized by the OSA integration module. For information about the OSA Explorer, see “The OSA Explorer” on page 452. Corresponds to: the osaexplorer command (see “Object Structure Awareness (OSA) Commands” in Chapter 15, “Scripting Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Debugger Notes	Opens a Note Browser, which allows you to browse your Debugger Notes. See “Viewing Debugger Notes” on page 157. Corresponds to: the noteview command (see Chapter 8, “Debugger Note Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Print Expression	Opens a dialog box for you to specify an expression to be printed. Corresponds to: the print command (see Chapter 9, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
View Expression	Opens a dialog box for you to enter an expression to view in a Data Explorer. Corresponds to: Shift + F9 and the view command (see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Menu Item	Meaning
List	Opens the List submenu. See “The List Submenu” on page 517.
Source Path	Opens the Source Path window, which allows you to change the directories that are searched for source files. Corresponds to: the source command (see “General Configuration Commands” in Chapter 7, “Configuration Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Refresh Views	Refreshes all Data Explorers, Register View windows, Memory View windows, Call Stack windows, etc. (If a Data Explorer variable is frozen, this menu item does not update that variable.) Corresponds to: the update command (see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Auto Update Views	Toggles whether the Debugger automatically updates all Data Explorers, Register View windows, Memory View windows, Call Stack windows, etc. (If a Data Explorer variable is frozen, this menu item does not update that variable.) The default update interval time is 1 second. To change the interval time, enter the update command followed by an interval number in seconds. For more information about the update command, see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
Close All Views	Closes all Data Explorers, Browse windows, Register View windows, Memory View windows, Call Stack windows, and Breakpoints windows. Corresponds to: the viewdel command (see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

The Navigation Submenu

The following table describes the selections available from the **View → Navigation** submenu.

Menu Item	Meaning
UpStack	Views the procedure one higher on the call stack. Corresponds to: , Ctrl + +, and the E + command (see Chapter 9, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
DownStack	Views the procedure one lower on the call stack. Corresponds to: , Ctrl + -, and the E - command (see Chapter 9, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Current PC	Views the procedure where the process is currently stopped. Corresponds to:  and the E command (see Chapter 9, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
UpStack To Source	Views the first procedure higher on the call stack that has source code. You can use this feature if you are stopped inside a library function with no source code (such as <code>printf()</code>), and you want to return to viewing your program. Corresponds to: the uptosource command (see Chapter 12, “Navigation Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Goto Location	Opens a dialog box in which you can specify a procedure or an address, and displays the program at the specified location in the source pane. Corresponds to: the e command (see Chapter 12, “Navigation Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

The Display Mode Submenu

The following table describes the selections available from the **View → Display Mode** submenu.

Menu Item	Meaning
Source Only	Displays only source code in the source pane. Corresponds to:  (not selected) and the assem off command (see Chapter 9, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Interlaced Assembly	Displays source code and the corresponding assembly instructions for each source code line in the source pane. Corresponds to:  (selected) and the assem on command (see Chapter 9, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Assembly Only	Displays only assembly instructions in the source pane. Corresponds to: the assem nosource command (see Chapter 9, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

The List Submenu

The following table describes the selections available from the **View → List** submenu.

See also the **l** (lowercase L) command in Chapter 9, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book.

Menu Item	Meaning
Files	Lists all the filenames in the program.
Procedures	Lists the names and addresses of all procedures.
Mangled Procedures	Lists the mangled names and addresses of all procedures.
Globals	Lists the names and addresses of all global variables.
Statics	Lists the names and addresses of all static variables.
Locals	Lists the names and values of all local variables for the procedure you are viewing (that is, the procedure at the current line pointer) if the procedure is on the stack.
Local Addresses	Lists the names and addresses of the local variables specified above.
Registers	Lists the names and values of all registers.

Menu Item	Meaning
Register Synonyms	Lists the register synonyms.
Variables In Procedure	Lists all the parameters and local variables of the specified procedure if it is on the stack.
Defines	Lists all the defined macros.
MULTI Variables	Lists the values of all MULTI special variables.
Processes	Lists the processes that the MULTI Debugger is currently attached to.
Signals	Lists the names and configuration settings of all signals.
Breakpoints	Lists all software breakpoints.
Dialog Boxes	Lists all loaded dialog boxes.
Source Paths	Lists the directories where MULTI looks for source files and scripts.
Included Files	Lists the source files included by the current file.

The Browse Menu

The following table describes the selections available from the **Browse** menu in the main Debugger window.

Menu Item	Meaning
Procedures	Opens a Browse window displaying all procedures in the program being debugged. See “Browsing Procedures” on page 200.
Globals	Opens a Browse window displaying all global variables in the program being debugged. See “Browsing Global Variables” on page 209.
Files	Opens a Browse window displaying the name and location of all of the source files containing procedures used in the program being debugged. See “Browsing Source Files” on page 211.
All Types	Opens a Browse window displaying all data types used in the program being debugged. See “Browsing Data Types” on page 214.
Classes	Opens a Tree Browser displaying the class hierarchy of the program being debugged. See “Browsing Classes” on page 224.

Menu Item	Meaning
Static Calls	Opens a Tree Browser displaying the static calling relationships of the current procedure (i.e., which functions are called by and which functions call the procedure at the current line pointer in the Debugger). See “Browsing Static Calls By Function” on page 225.
Dynamic Calls	Opens a Tree Browser displaying which functions were actually called during run time. See “Browsing Dynamic Calls by Function” on page 226.
File Calls	Opens a Tree Browser displaying static calls by file. This view shows source files whose functions are called from other source files. See “Browsing Static Calls By File” on page 226.
Includes	Opens a Graph View window displaying the include file hierarchy. See “Browsing Includes” on page 228.
Procedures In File	Opens a dialog box that allows you to specify a source file, then opens a Browse window displaying all the procedures in the file you specified. See “Browsing Procedures” on page 200.
Type	Opens a dialog box that allows you to specify a type name, then opens a Data Explorer displaying the structure of the type you specified. See Chapter 9, “Viewing and Modifying Variables with the Data Explorer”.

The Target Menu

The following table describes the selections available from the **Target** menu in the main Debugger window.

Menu Item	Meaning
Connect	Opens the Connection Chooser, which allows you to create, edit, or invoke a Connection Method to connect to your target. See “Standard Connection Methods” on page 57. Corresponds to: , F4, and the connect command (see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Menu Item	Meaning
Show Connection Organizer	Opens the Connection Organizer, which configures how you connect to target hardware or simulators. For more information about the Connection Organizer, see “Using the Connection Organizer” on page 75. Corresponds to: the connectionview command (see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Disconnect from Target	Disconnects from the current target debug server. Corresponds to:  and the disconnect command (see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Set Run-Mode Partner	Opens the Set Run-Mode Partner dialog box, in which you can select a run-mode connection that the Debugger automatically attempts to establish when you boot an INTEGRITY kernel via the current freeze-mode connection. The dialog box also allows you to disable this behavior. For more information, see “The Set Run-Mode Partner Dialog Box” on page 72.
Load Module	Opens a submenu that allows you to load a module into the target system’s memory. This menu item is only available if the target supports and was configured with a dynamic loader (for example, the LoaderTask on INTEGRITY).
Unload Module	Opens a submenu that allows you to unload a module.
1 connection 2 connection 3 connection 4 connection	Lists the most recently connected debug servers. To connect to one of them, select it.
IO Buffering	Toggles buffering for the I/O pane. Corresponds to: the iobuffer command (see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Menu Item	Meaning
Memory Manipulation	Opens the Memory Manipulation submenu. See “The Memory Manipulation Submenu” on page 521.
Memory Test	Opens the Memory Test Wizard . See “Quick Memory Testing: Using the Memory Test Wizard” on page 390.

The Memory Manipulation Submenu

The following table describes the selections available from the **Target** → **Memory Manipulation** submenu.

Menu Item	Meaning
Copy	Opens the Copy a Region of Memory dialog box, which allows you to copy memory contents from the From address to the Destination address in memory. The copying continues for the specified Number of blocks of memory, and you can specify the size of each memory block in bytes. Corresponds to: the copy command (see “General Memory Commands” in Chapter 11, “Memory Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Fill	Opens the Fill a Region of Memory dialog box, which allows you to fill memory starting from the Starting at address location with the given value Fill value. The filling continues for the specified Number of blocks of memory, and you can specify the size of each memory block in bytes. Corresponds to: the fill command (see “General Memory Commands” in Chapter 11, “Memory Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Find	Opens the Find a Region of Memory dialog box, which allows you to find a value in memory. The search begins at the Starting at address location and continues for Number of blocks of memory. You can specify the size of each memory block in bytes. For each memory location, MULTI performs a bitwise AND operation using the memory value and the mask as the operands, then compares the results with the Value to find. Corresponds to: the find command (see “General Memory Commands” in Chapter 11, “Memory Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Menu Item	Meaning
Compare	Opens the Compare Memory Regions dialog box, which allows you to compare memory contents. You specify the two starting memory locations to compare (Address 1 and Address 2) and the Number of blocks of memory to compare. You can specify the size of each memory block in bytes. You can also specify the Comparison operation: == (equal), > (greater than), >= (greater than or equal), < (less than), <= (less than or equal), != (not equal). Corresponds to: the compare command (see “General Memory Commands” in Chapter 11, “Memory Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Memory Load	Opens the Load Memory into Target dialog box, which allows you to copy a section of memory from the specified file (Load memory from file). Corresponds to: the memload command (see “General Memory Commands” in Chapter 11, “Memory Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Memory Dump	Opens the Dump Memory into File dialog box, which allows you to copy a section of memory to the specified file (Dump memory to file), starting at the Start Address and continuing for Length bytes. Corresponds to: the memdump command (see “General Memory Commands” in Chapter 11, “Memory Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

The Tools Menu

The following table describes the selections available from the **Tools** menu in the main Debugger window.

Menu Item	Meaning
MULTI EventAnalyzer	Opens the MULTI EventAnalyzer. This menu item is only available for some RTOSes. Corresponds to: 
MULTI ResourceAnalyzer	Opens the MULTI ResourceAnalyzer. This menu item is only available if you are using INTEGRITY. Corresponds to: 
Launcher	Opens the MULTI Launcher. See “The MULTI Launcher” on page 4. Corresponds to: the multibar command (see Chapter 3, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Project Manager	Opens the Project Manager on the project for the current program. If MULTI cannot find a project for the current program, it will open the Project Manager on a new project. Corresponds to: the builder command (see Chapter 5, “Building Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Rebuild	Rebuilds the current program if the project for the program can be located. Corresponds to: the build command (see Chapter 5, “Building Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Editor	Opens an Edit File dialog box, which allows you to select a file to open in an Editor window. Corresponds to:  and the edit command (see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Notepad	Opens an Editor window on a scratch file. Corresponds to: the note command (see Chapter 3, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Menu Item	Meaning
Launch Utility Programs	Opens the Utility Program Launcher , which provides a graphical interface to the Green Hills utility programs. Corresponds to: the wgutils command (see Chapter 5, “Building Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Serial Terminal	Opens the Serial Terminal submenu. See “The Serial Terminal Submenu” on page 524.
Search	Opens MULTI’s search dialog box. See “The Source Pane Search Dialog Box” on page 552. Corresponds to: the dialogsearch command (see Chapter 16, “Search Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Search in Files	Opens the Search in Files dialog box, which searches the source files of the program being debugged and any other open files for the specified string, using the grep utility. See “Searching in Files” on page 133. Corresponds to: the grep command (see Chapter 16, “Search Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

The Serial Terminal Submenu

The following table describes the selections available from the **Tools → Serial Terminal** submenu. For more information, see Chapter 20, “Establishing Serial Connections”.

Menu Item	Meaning
Make Serial Connection	Opens the Serial Connection Chooser , which allows you to create, edit, or invoke a Serial Connection Method. Corresponds to: the serialconnect command (see “Serial Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Menu Item	Meaning
Disconnect from Serial	Closes the current serial connection. Corresponds to: the serialdisconnect command (see “Serial Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
1 connection 2 connection 3 connection 4 connection	Lists the four most recently used serial connections. To invoke one of these, select it.

The Config Menu

The following table describes the selections available from the **Config** menu in the main Debugger window.

Menu Item	Meaning
Options	Opens the Options dialog box, which allows you to configure the appearance and behaviors of the Debugger and other MULTI tools. For a description of all the options in this window, see Chapter 9, “Configuration Options” in the <i>MULTI: Editing Files and Configuring the IDE</i> book. Corresponds to: the configoptions command (see “General Configuration Commands” in Chapter 7, “Configuration Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Customize Menus	Opens the Customize Menus window, which allows you to configure the menus in a number of MULTI tools. For more information, see Chapter 8, “Configuring and Customizing MULTI” in the <i>MULTI: Editing Files and Configuring the IDE</i> book. Corresponds to: the customizemenus command (see “Button, Menu, and Mouse Commands” in Chapter 7, “Configuration Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Menu Item	Meaning
Customize Toolbar	Opens the Customize Toolbar window, which allows you to configure the Debugger toolbar. For more information, see “Configuring the Debugger Toolbar” on page 538. Corresponds to: the customizetoolbar command (see “Button, Menu, and Mouse Commands” in Chapter 7, “Configuration Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Save Configuration as User Default	Saves the current configuration into the default user configuration file, so that it will be automatically loaded when MULTI starts. For more information, see “Saving Configuration Settings” in Chapter 8, “Configuring and Customizing MULTI” in the <i>MULTI: Editing Files and Configuring the IDE</i> book. Corresponds to: the saveconfig command (see “General Configuration Commands” in Chapter 7, “Configuration Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Clear User Default Configuration	Deletes the default user configuration file and restores the default system configuration. Corresponds to: the clearconfig command (see “General Configuration Commands” in Chapter 7, “Configuration Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Save Configuration As	Opens a file chooser in which you can choose the file where you want to save the current configuration. For more information, see “Saving Configuration Settings” in Chapter 8, “Configuring and Customizing MULTI” in the <i>MULTI: Editing Files and Configuring the IDE</i> book. Corresponds to: the saveconfigtofile command (see “General Configuration Commands” in Chapter 7, “Configuration Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Load Configuration	Opens a file chooser from which you can choose the file whose configuration you want to load. Corresponds to: the loadconfigfromfile command (see “General Configuration Commands” in Chapter 7, “Configuration Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Menu Item	Meaning
Set INTEGRITY Distribution	Opens the Default INTEGRITY Distribution dialog box in which you can provide MULTI with the location of the installed INTEGRITY distribution. This information is used to add INTEGRITY documentation to MULTI's Help menu and to determine the default INTEGRITY distribution for use with the Project Wizard. For more information, see "Using MULTI with INTEGRITY or u-velOSity" in Chapter 2, "Installation" in the <i>MULTI: Getting Started</i> book. Corresponds to: the setintegritydir command (see "General Configuration Commands" in Chapter 7, "Configuration Command Reference" in the <i>MULTI: Debugging Command Reference</i> book).
Set u-velOSity Distribution	Opens the Default u-velOSity Distribution dialog box in which you can provide MULTI with the location of the installed u-velOSity distribution. This information is used to add u-velOSity documentation to MULTI's Help menu and to determine the default u-velOSity distribution for use with the Project Wizard. For more information, see "Using MULTI with INTEGRITY or u-velOSity" in Chapter 2, "Installation" in the <i>MULTI: Getting Started</i> book. Corresponds to: the setuvelocitydir command (see "General Configuration Commands" in Chapter 7, "Configuration Command Reference" in the <i>MULTI: Debugging Command Reference</i> book).
State	Opens the State submenu. See "The State Submenu" on page 527.

The State Submenu

The following table describes the selections available from the **Config** → **State** submenu.

Menu Item	Meaning
Show Command History	Prints the command history. Each Debugger window keeps a history of all the Debugger commands entered in that window. You can use the ! command to execute a command from the history (see “History Commands” in Chapter 15, “Scripting Command Reference” in the <i>MULTI: Debugging Command Reference</i> book). You can also use the UpArrow and DownArrow keys to navigate through the history and choose a command for execution. Corresponds to: the h command (see “History Commands” in Chapter 15, “Scripting Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Save State	Saves the state of the Debugger to the specified file. The saved information includes target connection and status, breakpoints, and the source directories list. (Note that the Debugger may not be able to restore saved group breakpoints.) Corresponds to: the save command (see Chapter 3, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Restore State	Restores the state of the Debugger from the specified file. (Note that the Debugger may not be able to restore group breakpoints.) Corresponds to: the restore command (see Chapter 3, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Record Commands	Records commands into the specified file. Corresponds to: the > command (see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Record Commands + Output	Records commands and their output to the specified file. Corresponds to: the >> command (see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Stop Recording Commands	Stops recording commands. Use this item to stop recording if it has been started by Record Commands. Corresponds to: the >c command (see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Menu Item	Meaning
Stop Recording Commands + Output	Stops recording commands and their output. Use this item to stop recording if it has been started by Record Commands + Output. Corresponds to: the >>c command (see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Playback Commands	Plays back commands recorded in the selected file. Corresponds to: the < command (see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

The Windows Menu

The **Windows** menu is used to jump between all of the windows created by MULTI. If you choose an entry in the **Windows** menu, the corresponding window is brought to the front.

The following are menu items that may appear in the **Windows** menu:

Menu Item	Meaning
<i>debug window</i>	Gives focus to a debugging-related window that has been launched from the local executable.
<i>application name</i>	Gives focus to the main Debugger window for the local <i>application</i> .
<i>IDE tool</i>	Opens a submenu that lists windows corresponding to the <i>IDE_tool</i> . The windows listed in this submenu might be created from the local execution or from other executions. If only one MULTI Launcher is open, that tool does not have a submenu; it has a menu entry instead.
Misc	Opens a submenu listing other window types that do not fall into any of the above categories.

The Help Menu

The table below describes the selections available from the **Help** menu in the main Debugger window.

Menu Item	Meaning
Debugger Help	Opens online help for the Debugger. You may also press F1 to access the online help.
Manuals	Opens the Manuals submenu, which displays a list of all manuals included in your MULTI installation. Choose a manual to open its online help.
Bookmarks	Opens the Bookmarks submenu, which displays all the Help Viewer bookmarks you have created. Bookmarks function across manuals and MULTI sessions. Select a bookmark to display the bookmarked page in online help. Select Manage Bookmarks to open a window that allows you to modify your bookmarks.

Menu Item	Meaning
About MULTI	Opens the About the MULTI Debugger dialog box, which contains basic information about MULTI, such as its version and copyright information. Corresponds to: the about command (see Chapter 10, “Help and Information Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
License Info	Opens the Active Licenses dialog box, which displays the licenses in use.

The Debugger Window Toolbar

The Debugger toolbar contains buttons that allow you to access the most commonly used features of the Debugger. Placing the mouse pointer over a button on the toolbar displays a tooltip with a description of the button’s function. The following table gives a full description of each button and its command equivalent. For instructions about how to add or remove buttons from the toolbar, see “Adding, Removing, and Rearranging Toolbar Buttons” on page 538.

Button	Effect
 Go Back	Runs backward to the previous breakpoint or, if no previous breakpoint exists, to the first instruction in the trace data. Corresponds to: Alt + F5 and the bc command. For information about this command, see “Run Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Step Up	Steps back up to the caller of the current function. Corresponds to: Alt + F9 and the bcU command. For information about this command, see “Single-Stepping Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

Button	Effect
 Previous	Steps back one line. If the TimeMachine Debugger is in source display mode, clicking this button causes the TimeMachine Debugger to step back a single source line. If the TimeMachine Debugger is in assembly display mode, it steps back a single assembly instruction. For more information about display modes, see “Source Pane Display Modes” on page 41. Corresponds to: Alt + F10 and the bprev command. For information about this command, see “Single-Stepping Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Step Back	Steps back one line, stepping into a function if the previous line is in a different function. Corresponds to: Alt + F11 and the bs command. For information about this command, see “Single-Stepping Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Step (into Functions) on Selected Items	For items selected in the target list, executes one statement. If the statement is a function call, it steps into the called function. When in interlaced source/assembly mode, a machine instruction is executed instead of a source statement. Corresponds to: Debug → Step (into Functions) on Selected Items, F11, and the s command. For information about this command, see “Single-Stepping Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Next (over Functions) on Selected Items	For items selected in the target list, executes until the next statement of the current function (that is, steps over function calls). When in interlaced source/assembly mode, a machine instruction is executed instead of a source statement. Corresponds to: Debug → Next (over Functions) on Selected Items, F10, and the n command. For information about this command, see “Single-Stepping Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

Button	Effect
 Return on Selected Items	For items selected in the target list, continues to the end of the current function, and stops in the calling function after returning to it. Corresponds to: Debug → Return on Selected Items, F9, and the cU command. For information about this command, see “Continue Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Go on Selected Items	For items selected in the target list, begins execution of the program. If the process is stopped, this button continues execution. Corresponds to: Debug → Go on Selected Items, F5, and the c command. For information about this command, see “Continue Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Halt on Selected Items	For items selected in the target list, interrupts program execution. Corresponds to: Debug → Halt on Selected Items and the halt command. For information about this command, see “Halt Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Restart	Restarts the process with the same arguments as before. In TimeMachine mode, clicking this button restarts the process from the first traced instruction, but does not pass any arguments. If you click this button while debugging a Dynamic Download INTEGRITY application, MULTI attempts to (re)load the application. This button may not be available for relocatable modules. Corresponds to: Debug → Restart, Ctrl + Shift + F5, and the restart command. For information about this command, see “Run Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

Button	Effect
 Prepare Target	Automatically prepares your target or opens the Prepare Target dialog box, which allows you to download your program, flash your program, or verify that your program is present on the target. For more information, see “Preparing Your Target” on page 90. This button may not be available for relocatable modules. Corresponds to: Debug → Prepare Target and the prepare_target command. For information about this command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Reset	Resets the target board. This button is only available when you are connected to a target that supports the reset command. Corresponds to: the reset command. For information about this command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Reload Symbols	Reloads the current executable. Corresponds to: the debug command. For information about this command, see Chapter 3, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Assembly	Toggles between displaying source code only and source interlaced with assembly code. This button appears to be pushed down if any assembly instructions are displayed. Corresponds to: View → Display Mode → Source Only, View → Display Mode → Interlaced Assembly, and the assem command. For information about this command, see Chapter 9, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Program Counter	Displays the current program counter (PC) in the source pane. Corresponds to: View → Navigation → Current PC and the E command. For information about this command, see Chapter 9, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

Button	Effect
 Upstack	Displays the function up one stack frame. Hold this button down to view a menu showing the entire call stack. Corresponds to: View → Navigation → UpStack, Ctrl + +, and the E+ command. For information about this command, see Chapter 9, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Downstack	Displays the function down one stack frame. Hold this button down to view a menu showing the entire call stack. Corresponds to: View → Navigation → DownStack, Ctrl + –, and the E- command. For information about this command, see Chapter 9, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Call Stack	Displays the call stack in the Call Stack window. See also “Viewing Call Stacks” on page 376. Corresponds to: View → Call Stack and the callsvie command. For information about this command, see Chapter 6, “Call Stack Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Breakpoints	Opens a Breakpoints window in which you can add and edit breakpoints. See “Viewing Breakpoint and Tracepoint Information” on page 105. Corresponds to: View → Breakpoints and the breakpoints command. For information about this command, see Chapter 4, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Memory	Opens a Memory View window. See Chapter 13, “Using the Memory View Window”. Corresponds to: View → Memory and the memview command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

Button	Effect
 Registers	Opens a Register View window displaying machine registers. See Chapter 11, “Using the Register Explorer”. Corresponds to: View → Registers and the regview command. For information about this command, see “Register Commands” in Chapter 7, “Configuration Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Locals	Opens a Data Explorer displaying local variables. Corresponds to: View → Local Variables, the view \$locals\$ command, and the localsview command. For information about these commands, see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Connect	Connects to a target. If the Debugger is not connected, pressing this button opens the Connection Chooser dialog box, which allows you to connect to a target board or simulator. For more information about target connections, see the <i>MULTI: Configuring Connections</i> book for your target. Corresponds to: Target → Connect, F4, and the connect command. For information about this command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Disconnect	Disconnects from a target board or simulator. Corresponds to: Target → Disconnect from Target and the disconnect command. For information about this command, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Edit	Opens an Editor window on the currently active function. Corresponds to: Tools → Editor and the edit command. For information about this command, see “General View Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 OSA Explorer	Opens the OSA Explorer on the RTOS running on the target. This button is only available for some RTOSes.

Button	Effect
 Dump and Show Events	Dumps the event log and displays events in the MULTI EventAnalyzer. This button is only available for some RTOSes.
 MULTI EventAnalyzer	Launches the MULTI EventAnalyzer. This button is only available for some RTOSes. Corresponds to: Tools → MULTI EventAnalyzer
 MULTI ResourceAnalyzer	Launches the MULTI ResourceAnalyzer. This button is only available if you are using INTEGRITY. Corresponds to: Tools → MULTI ResourceAnalyzer
 INTEGRITY OSA Object Viewer	Launches the OSA Object Viewer on the object currently selected in the target list. If multiple items are selected in the target list, the OSA Object Viewer displays information for the entire INTEGRITY target. This button is visible only if you are debugging a run-mode connection and using INTEGRITY version 10 or later. Corresponds to: the osaview -context command if one item is selected in the target list. Corresponds to: the osaview command if multiple items are selected in the target list. For information about this command, see “Object Structure Awareness (OSA) Commands” in Chapter 15, “Scripting Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Close Debugger	Quits the MULTI Debugger. If you are debugging a process, MULTI gives you the choice to Quit and kill process or Detach from process . Whether this button appears on your toolbar depends on the setting of the option Display close (x) buttons . To access this option, select Config → Options → General Tab . Corresponds to: File → Close Debugger Window , Ctrl + Q, and the quit window command. For information about this command, see Chapter 3, “General Debugger Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

For information about the buttons that appear to the right of the **Status** column, see “Repositioning and Hiding the Target List” on page 33.

Configuring the Debugger Toolbar

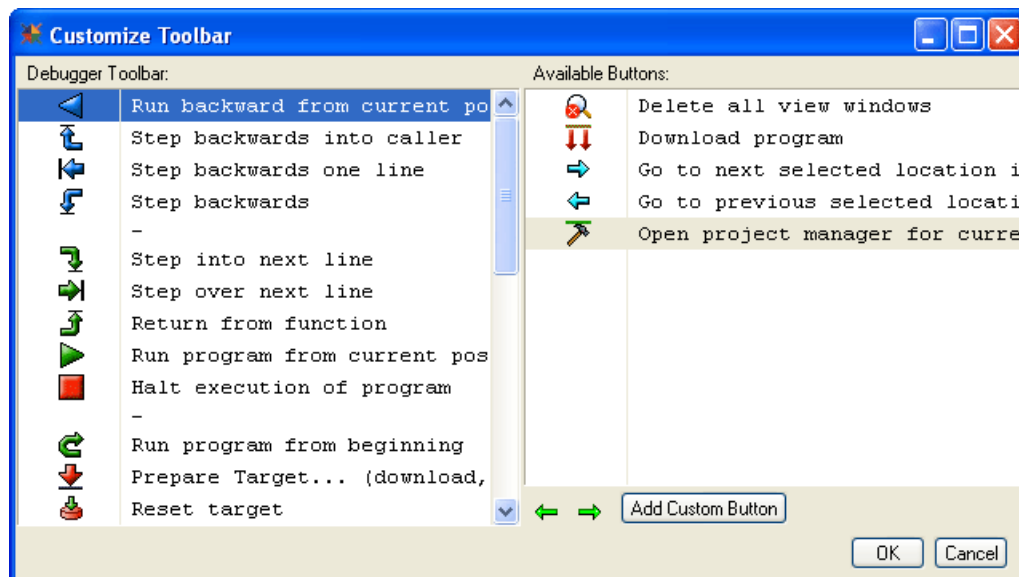
The toolbar appears just below the menu bar in the main Debugger window and displays the default buttons (documented in the previous section) with icon labels. As described next, you can add additional toolbar buttons or remove standard buttons, modify the placement of buttons, and configure buttons to display text labels instead of icons.

Adding, Removing, and Rearranging Toolbar Buttons

You can use the **Customize Toolbar** window to rearrange the order of buttons on the Debugger toolbar, add pre-defined and custom buttons, and delete buttons.

To open the **Customize Toolbar** window, do one of the following:

- Select **Config → Customize Toolbar**.
- In the Debugger command pane, enter the **customizetoolbar** command. For information about this command, see “Button, Menu, and Mouse Commands” in Chapter 7, “Configuration Command Reference” in the *MULTI: Debugging Command Reference* book.



The pane on the left side of the window—**Debugger Toolbar**—displays all the buttons that may currently appear on your toolbar. If your toolbar does not contain all the buttons located in the **Debugger Toolbar** pane, your current

debugging environment does not support them. For information about these buttons, see “The Debugger Window Toolbar” on page 531. The pane on the right side of the window—**Available Buttons**—displays a comprehensive set of the pre-defined buttons that you can add to your toolbar. For a description of these buttons, see the next section.

The following list provides instructions for using the **Customize Toolbar** window.

- To rearrange the order of buttons on the toolbar — In the **Debugger Toolbar** pane, drag a button to its new position.
- To delete a button from your toolbar — Select the button in the **Debugger Toolbar** pane and press the Delete key on your keyboard or click the **Remove button** button (➡).
- To add a pre-defined button to your toolbar — Drag the button from the right side of the window (**Available Buttons**) to the left (**Debugger Toolbar**), or select the button in the right side of the window and click the **Add new button** button (←). (You can add back deleted buttons in this way.)
- To add a custom button to your toolbar:
 1. Click **Add Custom Button**.
 2. In the dialog box that appears, select a button icon from the **Icon** pane. Fill in the **Button Name** text field with the text you want to appear in a tooltip when the cursor moves over the button, the **Command** text field with the command you want to execute when you click the button, and the **Help String** text field with the text you want to appear at the bottom of the Debugger window when the cursor moves over the button.
 3. Click **OK**.
 4. Your new button appears in the **Available Buttons** pane. Drag it to the desired location in the **Debugger Toolbar** pane.
 5. Click **OK**.

Modifications that you make to the toolbar through the **Customize Toolbar** window are saved by default.

You can edit the **button.odb** configuration file to make the same toolbar changes that are detailed above. The **button.odb** file is located in the following directory:

- Windows — *user_dir*\Application Data\GHS\v5
- UNIX — *user_dir*/.ghs/v5

Within the current MULTI session, you can reprogram any of the pre-defined Debugger toolbar buttons except the **Close Debugger** button (✗). You can define a button's name, command string, corresponding icon (optional), etc. by entering the **debugbutton** command with appropriate arguments. For more information about this command, see Chapter 8, "Configuring and Customizing MULTI" in the *MULTI: Editing Files and Configuring the IDE* book.

Pre-Defined Buttons

There are a number of buttons that do not appear on the Debugger toolbar by default, but that you can add via the **Customize Toolbar** window (see "Adding, Removing, and Rearranging Toolbar Buttons" on page 538). The following table contains descriptions of these buttons.

Button	Effect
 Delete all view windows	Closes all Data Explorers and Register View, Call Stack, Breakpoints, Memory View, and Browse windows. Corresponds to: View → Close All Views and the viewdel command. For information about this command, see "General View Commands" in Chapter 22, "View Command Reference" in the <i>MULTI: Debugging Command Reference</i> book.
 Download Program	Loads the current program into the target system's memory if the target supports and was configured with a dynamic loader (for example, the LoaderTask on INTEGRITY). Corresponds to: the load -setup command. For information about this command, see "General Target Connection Commands" in Chapter 18, "Target Connection Command Reference" in the <i>MULTI: Debugging Command Reference</i> book.
 Go to next selected location in trace data	Returns to the next location in the trace navigation history. This button is only available if you have previously clicked the Go to previous selected location in trace data button (described next). Corresponds to: the trace history + command. For information about this command, see Chapter 20, "Trace Command Reference" in the <i>MULTI: Debugging Command Reference</i> book.

Button	Effect
 Go to previous selected location in trace data	Returns to the previous location in the trace navigation history. This can be useful if you want to undo an action (such as running or stepping backwards in the TimeMachine Debugger) that brought you to an unexpected location in your source code. Corresponds to: the trace history - command. For information about this command, see Chapter 20, “Trace Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
 Open project manager for current project	Opens a Project Manager on the current project. Corresponds to: Tools → Project Manager and the builder command. For information about this command, see Chapter 5, “Building Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

The Target List Shortcut Menu

The following table describes the menu items that are available in the target list right-click menu. These menu items are context sensitive; they may not all appear every time you open this menu.

Menu Item	Meaning
Open in New Window	Opens a new Debugger window on the selected item. See also “Opening Multiple Debugger Windows” on page 32.
Prepare Target	Opens the Prepare Target dialog box from which you can choose to download the selected executable, flash it, or verify its presence on the target. For more information, see “The Prepare Target Dialog Box” on page 91.

Menu Item	Meaning
Use Connection	Opens a submenu with the following entries: • [Status] Connection_Name — Associates the current executable with the specified connection. Only compatible, currently active connections are listed. Status is one of: Available (the current executable can use this connection), Current (the current executable is using this connection), or Full (another executable is using this connection exclusively). • Stop Using Current Connection — Disassociates the selected executable from the connection it is currently associated with. Selecting this menu item does not disconnect from the target debug server. • Create New Connection — Opens the Connection Chooser and attempts to use the connection selected. For more information, see “Associating Your Executable with a Connection” on page 87.
Remove	Removes the currently selected executable from the target list and terminates the process. If the executable is part of the last remaining group of related items, the main Debugger window and its child windows close.
Disconnect from Target	Disconnects from the target debug server.
Resume	Starts or continues execution of the selected executable.
Halt	Halts the selected process.
Kill	Kills the selected process. If you are debugging a stand-alone program on a hardware target (i.e. not a simulated target), you cannot kill a running process. When the program reaches its exit point, the process terminates and the program is left halted at the exit system call.
Attach	Attaches the Debugger to the currently selected executable. This option is only available on some run-mode targets.

Source Pane Shortcut Menus

When you right-click in the source pane, a shortcut menu appears.



Note This is the default behavior. If you have configured a right-click to perform other functions, the shortcut menu will not appear.

This menu is context sensitive and depends on the object (such as a procedure, type, or variable) you right-click. Some menu items may appear dimmed, indicating that they are unavailable in the current context.

In the following shortcut menu discussion, the term “right-clicked line” refers to the source line where the right-click occurred.

Common Shortcut Menu Options

The following items appear in most shortcut menus available from the Debugger source pane.

Menu Item	Meaning
Display Program Visualizations	Opens a Graph View window (see “The Graph View Window” on page 227) displaying a graph with one tab for each defined custom <i>view description</i> . This is equivalent to the dataview command. For more information about how to define and load <i>view descriptions</i> , see Appendix D. For information about the dataview command, see “Data Visualization Commands” in Chapter 22, “View Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
Edit File	Opens an editor window centered on the right-clicked line of the current source file.
Search for Selected Text	Opens the Search in Files Results window, which displays search results for the text selected in the source pane.
Properties	Opens a dialog box displaying basic information about the current source line (for example, the filename, language, and line number), the current target connection, and also basic information about the right-clicked object (for example, its size, address, and whether its scope is static or global).

The Source Line Shortcut Menu

In addition to the menu options listed in “Common Shortcut Menu Options” on page 545, the following menu options are available in the shortcut menu that appears when you right-click a blank part of a source line or an unknown object.

Menu Item	Meaning
Run To This Line	Sets a temporary breakpoint on the line and then runs the program to it. The process stops on the line only if it attempts to execute that line. This menu item is only available if the right-clicked line contains executable code.
Change PC To This Line	Changes the program counter to the first executable instruction of the line. This menu item is only available if the right-clicked line contains executable code and the process is stopped within the procedure containing the right-clicked line.
Trace	Opens a submenu containing trace options. For a description of this submenu, see later in this section. This menu item is only available if you are connected to a trace-enabled target.

Menu Item	Meaning
Set Breakpoint	Opens a submenu that allows you to set various types of breakpoints on the line. For a description of this submenu, see later in this section. This menu item is only available if the right-clicked line contains executable code and does not already have a breakpoint set on it.
Edit Breakpoint	Opens the Software Breakpoint Editor , which allows you to edit the software breakpoint set on the line. This menu item is only available if a software breakpoint is set on the right-clicked line.
Remove Breakpoint	Removes the software breakpoint from the line. This menu item is only available if the right-clicked line contains executable code and a breakpoint has already been set on it.
Enable/Disable Breakpoint	Toggles (enables and disables) the software breakpoint located on the line. This menu item is only available if a breakpoint is set on the right-clicked line.
Move Breakpoint	Allows you to move the software breakpoint located on the line to another line in the same function. This menu item is only available if one software breakpoint is set on the right-clicked line. For more information, see “Moving Software Breakpoints” on page 109.
Set Hardware Breakpoint	Sets a hardware breakpoint on the line. This menu item is only available if the right-clicked line contains executable code and does not already have a breakpoint set on it.
Edit Hardware Breakpoint	Opens the Hardware Breakpoint Editor , which allows you to edit the hardware breakpoint set on the line. This menu item is only available if a hardware breakpoint is set on the right-clicked line.
Remove Hardware Breakpoint	Removes the hardware breakpoint from the line. This menu item is only available if the right-clicked line contains executable code and a hardware breakpoint has already been set on it.
Enable/Disable Hardware Breakpoint	Toggles (enables and disables) the hardware breakpoint located on the line. This menu item is only available if a hardware breakpoint is set on the right-clicked line.
Set Tracepoint	Opens the Tracepoint Editor , which allows you to set a tracepoint on the line.
Edit Tracepoint	Opens the Tracepoint Editor , which allows you to edit the tracepoint set on the line. This menu item is only available if a tracepoint is set on the right-clicked line.

Menu Item	Meaning
Remove Tracepoint	Removes the tracepoint from the line. This menu item is only available if a tracepoint is set on the right-clicked line.
Enable/Disable Tracepoint	Toggles (enables and disables) the tracepoint located on the line. This menu item is only available if a tracepoint is set on the right-clicked line.
Create Note	Allows you to add a Debugger Note at the line's location. For more information, see Chapter 8, "Using Debugger Notes".
Edit Note	Allows you to edit the Debugger Note set at the line's location. For more information, see Chapter 8, "Using Debugger Notes".
Remove Note	Removes the Debugger Note from the line. For more information, see Chapter 8, "Using Debugger Notes".
Browse Includers Of This File	Opens a Browse window displaying those files that include the current file. For more information, see "Browsing Source Files General Information" on page 212.
Browse Files Included By This File	Opens a Browse window displaying those files that are included by the current file. For more information, see "Browsing Source Files General Information" on page 212 .
Display Interlaced Assembly/Source Only	Toggles the display between source-only mode and interlaced assembly mode.

The following table describes the selections available from the **Set Breakpoint** submenu.

Menu Item	Meaning
Set Breakpoint	Sets a software breakpoint. See also the b command in Chapter 4, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book
Set And Edit	Opens the Software Breakpoint Editor window, which allows you to specify and set a software breakpoint. See the b command in Chapter 4, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book, and also “Creating and Editing Software Breakpoints” on page 106.
Set Jump Breakpoint	Queries for a location to jump to when the breakpoint is hit and sets a jump breakpoint. A jump breakpoint executes the g location;resume; commands when it is reached. For information about the g command, see “General Program Execution Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book. For information about the resume command, see Chapter 4, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.
Set Any Task Breakpoint	Sets a software breakpoint which can be hit by any task. This option is available only when connected to a target which supports this type of breakpoint. See the sb command in Chapter 4, “Breakpoint Command Reference” in the <i>MULTI: Debugging Command Reference</i> book.

The Procedure Shortcut Menu

In addition to the menu options listed in “Common Shortcut Menu Options” on page 545, the following menu options are available in the shortcut menu that appears when you right-click a procedure.

Menu Item	Meaning
Go To Definition	Displays the source code, if available, for the procedure’s definition in the source pane.
Go To Declaration	Displays the source code, if available, for the procedure’s declaration in the source pane.

Menu Item	Meaning
Browse References	Shows the procedure's cross references in a Browse window. For more information, see "Browsing Cross References" on page 216.
Browse Other	For a description of this submenu, see later in this section.
Trace	This is available only if connected to a trace-enabled target. For a description of this submenu, see later in this section.
Step Into This Function	Attempts to step through the next source line, halting if execution enters the selected procedure. This is available only when right-clicking procedure calls on the currently executing line.
Edit Definition	Opens an editor window centered on the procedure's definition, if the appropriate source debugging information is available.

The following table lists the entries in the **Browse Other** submenu.

Menu Item	Meaning
Browse Callers	Opens a Browse window displaying the callers of the procedure.
Browse Callees	Opens a Browse window displaying the callees of the procedure.
Browse Static Calls	Opens a Tree Browser window displaying the static call graph of the procedure (which shows its callers and callees and their callers and callees). See "The Tree Browser Window" on page 220.

The Variable Shortcut Menu

In addition to the menu options listed in "Common Shortcut Menu Options" on page 545, the following menu options are available in the shortcut menu that appears when you right-click a variable (local, parameter, global, or static).

Menu Item	Meaning
Go To Definition	Displays the source code, if available, for the variable's definition in the source pane.
Go To Declaration	Displays the source code, if available, for the variable's declaration in the source pane. This is available only for global and static variables.

Menu Item	Meaning
Browse References	Shows the variable's cross references in a Browse window. See "Browsing Cross References" on page 216 for more information.
View Value	Opens a Data Explorer displaying the value of the variable.
Explore Data	If the variable matches a custom data description of the data type <code>graph</code> , opens a Graph View window displaying a graph of objects based on the data description. See Appendix D and "The Graph View Window" on page 227.
Trace	This is available only if you are connected to a trace-enabled target. For a description of this submenu, see later in this section.
Edit Definition	Opens an editor window centered on the variable's definition, if the appropriate source debugging information is available.

The Type Shortcut Menu

In addition to the menu options listed in "Common Shortcut Menu Options" on page 545, the following menu options are available in the shortcut menu that appears when you right-click a type.

Menu Item	Meaning
Go To Definition	Displays the source code, if available, for the type's definition in the source pane.
Browse References	Shows the type's cross references in a Browse window. See "Browsing Cross References" on page 216 for more information.
Browse Other	For a description of this submenu, see later in this section. This is available only if the type is a C++ class.
View Type	Opens a Data Explorer displaying the type's definition.
Edit Definition	Opens an editor window centered on the type's definition, if the appropriate source debugging information is available.

The following table lists the entries in the **Browse Other** submenu.

Menu Item	Meaning
Browse Superclasses	Shows the type's super-classes in a Browse window.

Menu Item	Meaning
Browse Subclasses	Shows the type's sub-classes in a Browse window.
Browse Class Hierarchy	Opens a tree browser window to show class hierarchy information. See "The Tree Browser Window" on page 220.

The Type Member Shortcut Menu

In addition to the menu options listed in "Common Shortcut Menu Options" on page 545, the following menu options are available in the shortcut menu that appears when you right-click a type's member (such as the member reference of a struct variable, an enumeration member, or a member name in a class declaration).

Menu Item	Meaning
Go To Definition	Displays the source code, if available, for the member's definition in the source pane.
Browse Member References	Shows the member's cross references in a Browse window. See "Browsing Cross References" on page 216 for more information.
View Value	Opens a Data Explorer displaying the value of the member.
Explore Data	If the member variable matches a custom data description of the data type <code>graph</code> , opens a Graph View window displaying a graph of objects based on the data description. See Appendix D and "The Graph View Window" on page 227.
Edit Definition	Opens an editor window centered on the member's definition, if the appropriate source debugging information is available.

The C/C++ Macro Shortcut Menu

In addition to the menu options listed in "Common Shortcut Menu Options" on page 545, the following menu options are available in the shortcut menu that appears when you right-click a C or C++ macro.

Menu Item	Meaning
Go To Definition	Displays the source code, if available, for the macro's definition in the source pane.

The Command Pane Shortcut Menu

The following items appear in the shortcut menu when you right-click in the Debugger command pane (or the target pane, I/O pane, serial terminal pane, Python pane, or traffic pane).

Menu Item	Meaning
Cut	Cuts the current selection to the clipboard. Note that only the current input line may be cut.
Copy	Copies the current selection to the clipboard.
Paste	Pastes text from the clipboard into the current pane. The text will be treated as input.
Command Pane	Switches to the command pane.
Target Pane	Switches to the target pane.
IO Pane	Switches to the I/O pane.
Serial Pane	Switches to the serial terminal pane.
Python Pane	Switches to the Python pane.
Traffic Pane	Switches to the traffic pane.
Interleaved Output	When selected, output from each pane is displayed in the command pane in addition to its own pane.
Clear Pane	Clears all text from the current pane.
Show Separate Py Window	Opens the Python pane as a separate window.
Save As	Opens a file chooser in which you can choose the file where you want to save the contents of the current pane.

The Source Pane Search Dialog Box

To control searches in the source pane, open the **Source Pane Search** dialog box in one of the following ways:

- Select **Tools** → **Search**.
- In the command pane, enter the **dialogsearch** command. For information about this command, see Chapter 16, “Search Command Reference” in the *MULTI: Debugging Command Reference* book.

For example, to search for a string such as `tree`:

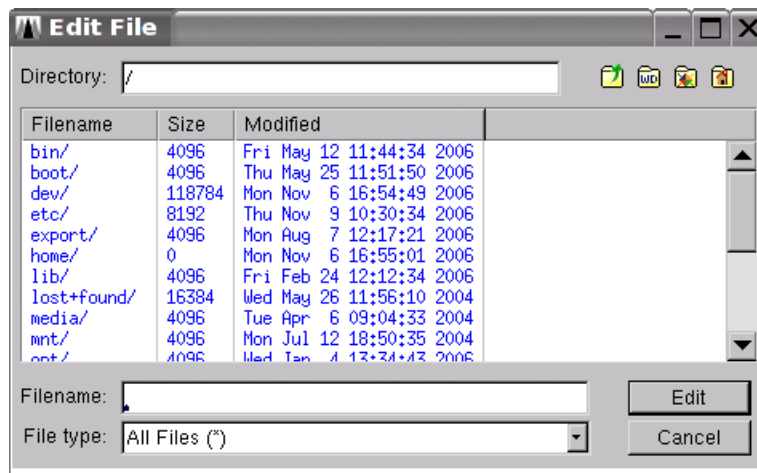
1. Type `tree` in the search text field.
2. Click **Find** to search for the next occurrence of the string.

The toggle buttons in this window are the same as those in the Editor's search dialog box.

To initiate or repeat searches without the search dialog box, you can also use the Ctrl+F or Ctrl+B key sequences in the command pane.

The File Chooser Dialog Box (UNIX)

A file chooser dialog box opens if you initiate an action (using a menu, button, command, or shortcut) for which a file must be specified, but you have not yet specified a file. A chooser also opens if you click a **Browse** or  button on a dialog box. A sample UNIX file chooser follows. (For information about Windows file choosers, see your Windows documentation.)



The title bar of the file chooser will vary depending on the action you are performing. The following table describes the main elements of the file chooser.

Item	Meaning
Directory	Displays the directory whose contents are shown in the File List . Type in a new directory name and press Enter to display a different directory.
Directory Buttons	With this set of buttons, you can quickly go to different important directories. The buttons that appear are:  — Up One Directory from the current directory.  — Jumps to the current Working Directory.  — Jumps to the MULTI Run Directory.  — Jumps to the User Home Directory.

Item	Meaning
File List	Below the directory text field is the file list. To enter a directory, double-click the directory. To choose a file, single-click the filename. You can click any column header to sort the list in ascending or descending order. If multiple files are allowed for the present operation (for example, Open is selected in the editor menu), use the Shift key to select a consecutive list of files; use the Ctrl key to select non-consecutive files.
Filename	Type a filename or directory name into this text field. As you type in this field, the selection in the file list will change to reflect the closest match. If you type a directory name and press Enter, or follow the directory name by a slash (/), the file list will change to the specified directory.
File type	If you select a file type, the File List will only display files with suffixes that match the selected file type.
Action buttons	There are two buttons in the lower right corner of the file chooser window. The upper button displays the action that will occur, such as Open (in the example above), OK, or Edit. The lower button is the Cancel button, which closes the window without taking any action.

Keyboard Shortcut Reference

Appendix B

This Appendix Contains:

- Main Debugger Window Shortcuts
- Source Pane Shortcuts
- Command Pane Shortcuts

This appendix contains descriptions of the Debugger's default keyboard shortcuts.

Main Debugger Window Shortcuts

The following shortcuts are available from the Debugger window.



Note If you have clicked in the source pane or command pane, these shortcuts may have different effects. See “Source Pane Shortcuts” on page 560 and “Command Pane Shortcuts” on page 561.

Shortcut	Effect
F1	Displays help about the current window. For more information about online help, see “Accessing Online Help” on page 8. Corresponds to: the help command (see Chapter 10, “Help and Information Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
F4	Connects to a target. If the Debugger is not connected, pressing this button opens the Connection Chooser dialog box, which allows you to connect to a target board or simulator. For more information about target connections, see the <i>MULTI: Configuring Connections</i> book for your target. Corresponds to:  and the connect command (see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
F5	Begins execution of the program. If the process is stopped, it continues execution. Corresponds to:  and the c command (see “Continue Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Shortcut	Effect
Ctrl + Shift + F5	Restarts the process with the same arguments as before. In TimeMachine mode, this shortcut restarts the process from the first traced instruction, but does not pass any arguments. If you use this shortcut while debugging a Dynamic Download INTEGRITY application, MULTI attempts to (re)load the application. This shortcut may not be available for use with relocatable modules. Corresponds to:  and the restart command (see “Run Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
F9	Continues to the end of the current procedure, and stops in the calling procedure after returning to it. Corresponds to:  and the cU command (see “Continue Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
F10	Executes until the next statement of the current procedure (that is, steps over procedure calls). When in interlaced source/assembly mode, a machine instruction is executed instead of a source statement. Corresponds to:  and the n command (see “Single-Stepping Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
F11	Executes one statement. If the statement is a procedure call, it is stepped into. When in interlaced source/assembly mode, a machine instruction is executed instead of a source statement. Corresponds to:  and the s command (see “Single-Stepping Commands” in Chapter 14, “Program Execution Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Shortcut	Effect
Ctrl + +	Displays the procedure up one stack frame. Corresponds to:  and the E + command (see Chapter 9, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).
Ctrl + –	Displays the procedure down one stack frame. Corresponds to:  and the E - command (see Chapter 9, “Display and Print Command Reference” in the <i>MULTI: Debugging Command Reference</i> book).

Source Pane Shortcuts

The following shortcuts are available from the Debugger window’s source pane.

Shortcut	Effect
PageUp	Scroll the source pane up by one page.
PageDown	Scroll the source pane down by one page.
Shift + UpArrow	Scroll the source pane up by one line.
Shift + DownArrow	Scroll the source pane down by one line.
Ctrl + F	Search forward in the source pane.
Ctrl + B	Search backward in the source pane.
Esc	Abort current operation or halt process.

Command Pane Shortcuts

The following shortcuts are available from the Debugger window's command pane.

Shortcut	Effect
UpArrow	If nothing is entered on the command line, retrieves the previous command from the command history, moving back in the list. Press repeatedly to retrieve older commands. If you have already begun typing, MULTI attempts to auto-complete the current string, working backwards through commands that have been entered.
DownArrow	If nothing is entered on the command line, retrieves the next command from the command history, moving forward in the list. Press repeatedly to retrieve more recent commands. If you have already begun typing, MULTI attempts to auto-complete the current string, working forwards through commands that have been entered.
Ctrl + UpArrow	Scrolls up four lines.
Ctrl + DownArrow	Scrolls down four lines.
Ctrl + U	Cuts to the beginning of the current line or the selection.
Tab	Accepts auto-completion of the current word.
Ctrl + P	Attempts to auto-complete the current string, working backwards through the command history for commands beginning with the typed characters.
Ctrl + N	Attempts to auto-complete the current string, working forwards through the command history for commands beginning with the typed characters.
Ctrl + D	Displays a list of possible completions for the current word.
F6	Cycles between the command, I/O, target, and serial terminal panes. For more information, see "The Cmd, Trg, I/O, Srl, Py, and Tfc Panes" on page 44 .
F12	Displays a pop-up menu showing all Debugger Notes set in the program.
Select text with the left mouse button	On UNIX, copies a string.
Click with the right mouse button	Opens a shortcut menu for the command pane. For a full description of the shortcut menu, see "The Command Pane Shortcut Menu" on page 552.

Command Line Reference

Appendix C

The following command line options can be used when starting MULTI from the command line, as described in “Using the Command Line” on page 22.

-- args

Passes all the arguments that follow the double dash to the program being debugged as though the arguments were set with the **setargs** command. This option is particularly useful for native debugging.

For more information, see the **setargs** command in “General Program Execution Commands” in Chapter 14, “Program Execution Command Reference” in the *MULTI: Debugging Command Reference* book.

-build

Immediately begins a build.

-c file**-config file (Windows only)****-configure file**

Configures MULTI using the information in the configuration file *file*. The **-config file** option is for Windows only. The other options apply to Windows and UNIX.

For more information, see “Creating and Editing Configuration Files” in Chapter 8, “Configuring and Customizing MULTI” in the *MULTI: Editing Files and Configuring the IDE* book.

-cmd commands

Runs the specified commands when the Debugger is up.

-connect[=*target* | =*connection_method_name*]

Connects to the target debug server specified by *target*; connects using the specified Connection Method; or, if neither a target nor a Connection Method is specified, opens the **Connection Chooser**.

For more information about connecting to targets, see Chapter 4, “Connecting to Your Target”. For information about the **connect** command, which corresponds to this option, see “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.

Note: The Debugger ignores the deprecated `mode` argument in Connection Methods that specify it. To remove the `mode` argument from a MULTI 4 Connection Method, edit and save the Connection Method in MULTI 5. For more information, see the **connect** command in “General Target Connection Commands” in Chapter 18, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book.

-connectfile file

Specifies that connections should be loaded from the file *file*.

For more information about loading connections from a file, see Chapter 4, “Connecting to Your Target”.

-D

Ignores all currently specified alternative directories.

See also the **-I** command line option later in this table and the **source** command in “General Configuration Commands” in Chapter 7, “Configuration Command Reference” in the *MULTI: Debugging Command Reference* book.

-data offset

Sets the default offset for all text addresses to *offset*, which is assumed to be in decimal unless preceded by `0x`. This option is only useful when debugging programs built with position-independent data; it should not be used in other kinds of programs.

-display disp**UNIX only**

Specifies that the Debugger will open in the alternative display *disp*.

-e entry

Specifies the entry label. The default is `main`. In C++ mode, the entry must be specified in such a way that it may be demangled.

-E file

Opens the Debugger on multiple files. Use this option for each file after the first that you want to debug. For example, if you want to debug **foo**, **bar**, and **baz**, use the command:

```
multi foo -E bar -E baz
```

This opens a single Debugger window and lists **foo**, **bar**, and **baz** in the target list of that window.

The maximum number of files you can specify is 256. The first file you specify, which is not preceded by **-E**, is included in this maximum number.

-h

Displays a concise description of all command line options. This is equivalent to the **-usage** option (below).

-H**-help**

Opens the MULTI Help Viewer on the *MULTI: Debugging* book.

-I directory

(This option is a capital letter “i.”)

Names an alternative directory for the Debugger to search for files in. You can specify multiple alternative directories by using multiple **-I directory** arguments. The Debugger searches alternative directories in the order given. If it does not find a file in the specified alternative directory or directories, it also searches the current directory.

See also the **-D** command line option earlier in this table and the **source** command in “General Configuration Commands” in Chapter 7, “Configuration Command Reference” in the *MULTI: Debugging Command Reference* book.

-m file

Sets *file* as the default specification file.

For more information, see “Using a Specification File” on page 25.

-nocfg

Causes MULTI to ignore all **.cfg** files on startup.

-nodisplay**UNIX only**

Specifies that the Debugger will open in non-GUI mode.

For more information, see “Starting in Non-Graphical Mode (UNIX only)” on page 21.

-norc

Causes MULTI to ignore **.rc** files on startup.

For more information, see “Using Script Files” in Chapter 8, “Configuring and Customizing MULTI” in the *MULTI: Editing Files and Configuring the IDE* book.

-noshared

Indicates that MULTI should not debug shared libraries. This option is relevant only for targets that support shared libraries.

See also the `DEBUGSHARED` system variable in “System Variables” on page 291.

-nosplash

Prevents MULTI from displaying the startup banner.

-notifylib library

Sets the name of the notification library to *library*.

-osa *osa_name* [#cfg=*configuration_file*] [#lib=*library_name*] [#log=*log_file*]

Starts MULTI with freeze-mode debugging and OS-awareness enabled. *osa_name* specifies an object structure-aware debug package. *configuration_file* specifies the configuration file for freeze-mode debugging and OS-awareness, *library_name* specifies the library containing the OSA integration package, and *log_file* specifies a file in which to log the interaction between MULTI and the OSA integration package.

For more information, see “Starting MULTI for Freeze-Mode Debugging and OS-Awareness” on page 451.

-p *file*

Runs the commands in the playback file *file* on startup.

For more information, see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

-P *pid***Native targets only**

Attaches to the process with the process identification number *pid*. The maximum number of processes you can specify is 256.

-preload *module*

Downloads *module* after connecting to an operating system target, which must be specified elsewhere on the command line.

This option is only valid for certain operating system targets. Additionally, the target must support and be configured with a dynamic loader (for example, the LoaderTask on INTEGRITY).

-r *file*

Records commands to the playback file *file*.

For more information, see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

-R *file*

Records commands and output to the playback file *file*.

For more information, see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

-rc file

Reads the command script *file* when the first Debugger window appears. The file is read after the global and user script files.

For more information, see “Using Script Files” in Chapter 8, “Configuring and Customizing MULTI” in the *MULTI: Editing Files and Configuring the IDE* book.

-RO file

Records output to the playback file *file*. (Commands are not recorded.)

For more information, see “Record and Playback Commands” in Chapter 15, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.

-run debug_server [-timeout=num] --

Connects to *debug_server*, runs *program* in the Debugger (see “Using the Command Line” on page 22), runs any commands specified on the command line or in any relevant *.rc* scripts, and then exits.

-timeout specifies the number of seconds until the operation times out. On Windows and Linux, the default is 600; on Solaris, the default is 2400. The maximum value of *num* is $2^{31}/1000$. If *num* is -1, the operation never times out.

When the Debugger exits, it returns exit code 0 (success) on completion even if the program running on the target returns a different exit code. If you need to check the exit code of an embedded target program, consider using the **grun** utility program. For information about **grun**, see the *MULTI: Building Applications* book for your target processor family.

This option should be specified after other options and before *program* (see “Using the Command Line” on page 22).

-servertimeout time

Sets the default debug server timeout to *time* seconds.

-socket port

Creates a socket connection on the port *port*. This socket connection can be used to send Debugger commands and receive Debugger output.

For more details, see the **socket** command in Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

-text offset

Sets the default offset for all text addresses to *offset*, which is assumed to be in decimal unless preceded by 0x. This option is only useful when debugging programs built with position-independent code; it should not be used with other kinds of programs.

-top**Native targets only**

Opens the Process Viewer, which displays a snapshot of the processes running on your native target and allows you to attach to native processes. For more information, see “Viewing Native Processes” on page 378.

-tv file

Specifies the task view configuration file for run-mode debugging.

For more information about task view configuration files, see “Task Group Configuration File” on page 444.

-usage

Displays a concise description of all command line options. This is equivalent to the **-h** option (above).

-V

Prints Debugger version information.

-wmhints**UNIX only**

Sends extra messages to older or non-compliant X Window managers to fix window display problems.

Creating Custom Data Visualizations

Appendix D

This Appendix Contains:

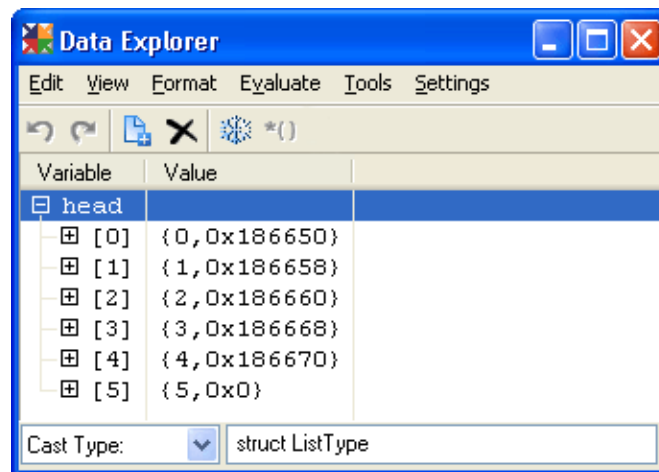
- Using MULTI Data Visualization (.mdv) Files
- Invoking Customized Data Visualizations
- MULTI Data Visualization (.mdv) File Format

MULTI data visualization (**.mdv**) files can be used to create customized views which display certain types of variables according to your specifications. Depending upon the types of data you want to display, and your specific settings, these custom views will be shown in the Data Explorer or **Graph View** windows. The following two examples use custom visualizations to define the way data types are displayed.

A linked list using the following structure definition in your source code:

```
struct ListType {  
    int value;  
    struct ListType* next;  
}
```

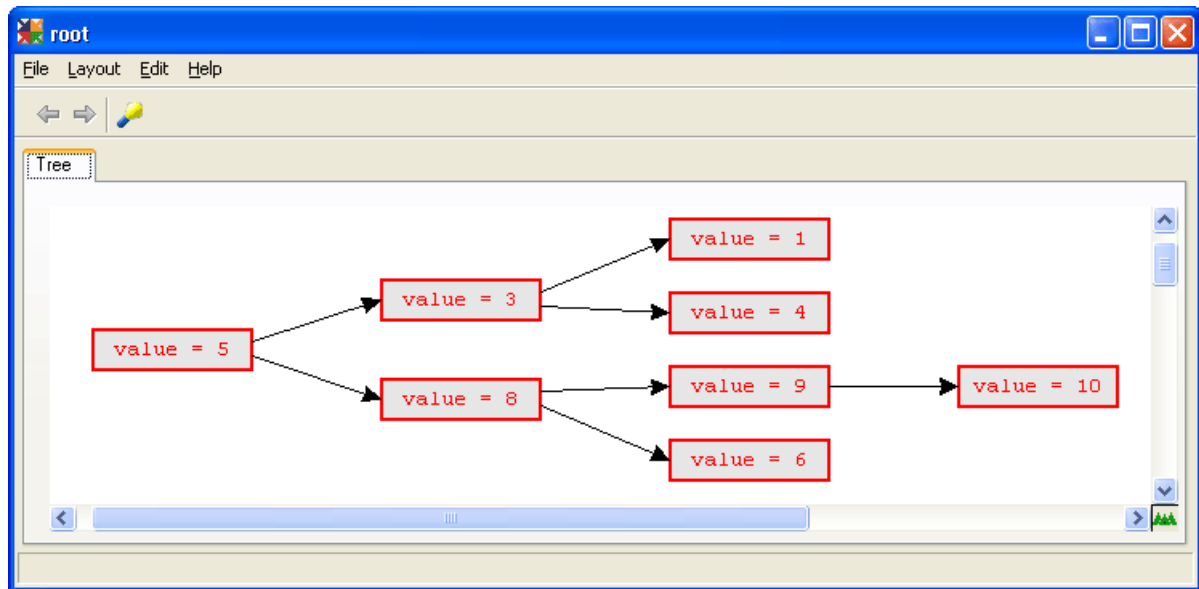
could be shown in a Data Explorer as:



A binary tree using the following structure definition in your source code:

```
struct TreeNode {  
    int value;  
    struct TreeNode *left, *right;  
}
```

could be shown in a **Graph View** window as:



Using MULTI Data Visualization (.mdv) Files

You can define a custom data visualization for any data type. (MULTI has built-in support for certain types of data structures. For information about using MULTI's built-in visualizations, see “Displaying Linked Lists” on page 171.) To use customized data visualizations, you must first create one or more MULTI data visualization (**.mdv**) files and then load them during your debugging session. These files can contain three kinds of descriptions that are used to create custom data visualizations:

- *Data descriptions* are the basic building blocks of custom data visualizations. They describe how MULTI should display variables of a particular type.
- *Profile descriptions* contain a collection of data descriptions, as well as some basic information about hierarchical relationships and the layout of the visualization(s) for the variables specified in the data descriptions.

You can create multiple profiles, but only one profile can be *active* at a time. Custom visualizations are only available for the variables described in the active profile or in no profile. (Data descriptions that are not included in a profile are considered to be part of a global profile that is always accessible.) You can change profiles easily by using the **dvprofile** *prof_name* command. For information about the **dvprofile** command, see “Data Visualization Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

- *View descriptions* specify a configuration of profiles to use for displaying variables in a **Graph View** window. These allow you to specify a profile to be active during evaluation of the view. They also allow you to create a “dual pane” display, where selecting an object in the first pane causes a new graph rooted on that object to be displayed in the second pane, using a different profile and therefore potentially providing a different view of the data.

Each of these potential components of **.mdv** files is described in more detail in “MULTI Data Visualization (.mdv) File Format” on page 578, but the following outline of the basic syntax will provide a context for understanding the descriptions. (Lines enclosed in square brackets in the following template are optional; the square brackets are not part of the syntax. The curly brackets, however, are part of the syntax.)


```
unique_prof_id {
  profile_name = prof_name
  [parent_profile = parent_prof_name]
  [default_root = def_root_prof]
  [vertical_layout = bool_expr]
  [add_siblings_of_root = bool_expr]

  data_descriptions {
    data_desc_name_1 {
      signature = {"sig1", "sig2", ... }
      [required_fields = {"field1", "field2", ... }]
      type = "data_desc_type"
      [predicate = "pred"]
      [required and optional fields depending on type] ...
    }
    ...
  }
}

unique_view_id {
  initial_pane = "pane_name"
  default_root = "def_root_view"
  [tab_name = "tab"]

  panes {
    pane_name1 {
      profile_name = "prof_name1"
      [child_pane = "child_pane_name1"]
    }
    ...
  }
}
```

For an example **.mdv** file, see “.mdv File Examples” on page 605.

Loading and Clearing MULTI Data Visualization (.mdv) Files

To make the data descriptions, profiles, and views that you have defined available in the Debugger, you must load your **.mdv** file(s) by entering the following command in the Debugger command pane:

dvload *filename*

where *filename* is the name of a custom MULTI data visualization (**.mdv**) file. You can load multiple **.mdv** files by issuing this command repeatedly, once for each file you want to load. MULTI will search the user configuration directory and installation configuration directory for the file if the file is specified with a relative path.

All of the data descriptions, profile descriptions, and view descriptions included in all loaded **.mdv** files will be available during your MULTI session, but will not persist across re-launches. However, only those data descriptions contained in the active profile or in no profile will be accessible for viewing, and only one profile can be active at a time. These are the data descriptions that MULTI will match your program data against when selecting data to be displayed in your custom view. (You can change the active profile to any profile in a loaded **.mdv** file by issuing the **dvprofile** *prof_name* command. See “Profile Descriptions” on page 600.)

To clear MULTI data visualization files you have loaded, use the following command:

dvclear

This command clears all loaded data descriptions.

Invoking Customized Data Visualizations

Most types of data descriptions are used to describe how a type should be displayed in Data Explorers and by the **print** command (see Chapter 9, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book). These data descriptions are used automatically when a variable matching the data description is viewed in a Data Explorer, or printed in the Debugger command pane.

The graph data description type, on the other hand, describes how to display a graphical representation of the variable in a **Graph View** window (see “The Graph View Window” on page 227). Data descriptions with the `use_for_print_and_view` field set to `false` are also not used when viewing a variable in a Data Explorer or printing to the Debugger command pane. They are typically used to define containers that are used in graph data description types. For example, the children of a node can be specified in a container, which would need a corresponding data description.

For information about how MULTI matches a data description to a variable, see “Data Descriptions” on page 578.

To view the graphical representation of data for which you have graph data descriptions, use one of the following methods:

- Right-click a variable and select **Explore Data** from the shortcut menu that appears. If you have right-clicked a variable matching a graph data description, MULTI will create a graph of objects based on the data description, and display it in a **Graph View** window. The graph will be rooted on the clicked variable. If no data description is found for the variable’s type, the text of the node will be the output of **print variable**.
- Right-click in the Debugger source pane and select **Display Program Visualizations** or enter the command **dataview** in the Debugger command pane (see “Data Visualization Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book). This will display a graph with one tab for each defined custom *view description*, using the default root and profile specified for each view.
- Enter the following command in the Debugger command pane:

dataview *prof_name* | *view_name*

The will open a graph displaying the specified profile (*prof_name*) or view (*view_name*). The graph will use default root (*def_root_prof* or *def_root_view*) specified in the profile or view description.

MULTI Data Visualization (.mdv) File Format

A MULTI data visualization (**.mdv**) can contain data descriptions, profile descriptions, and/or view descriptions. MULTI uses the information in these descriptions to filter which data to display and to format custom visualizations of the filtered data according to your specifications. The sections below give the exact syntax for each of these types of descriptions.



Note Many of the fields used to describe data types, profiles, and views in **.mdv** files take expressions. See “Using Expressions in MULTI Data Visualization (.mdv) Files” on page 604 for information about valid expressions.

Data Descriptions

Data descriptions describe how data is accessed for a particular type. A data description has the following format (lines enclosed in square brackets in the template below are optional; the square brackets are not part of the syntax, but the curly brackets are):

```
data_desc_name {  
    signature = {"sig1", "sig2", ... }  
    [required_fields = {"field1", "field2", ... }]  
    type = "data_desc_type"  
    [predicate = "pred"]  
    [required and optional fields depending on type]  
}
```

The meaning of the elements in this format are described in the following table.

`data_desc_name`

Uniquely identifies the data description. Data descriptions with the same name will overwrite each other.

`signature = {"sig1", "sig2", ... }`

Specifies the signature or signatures (*sig1*, *sig2*, etc.) of the data type being described in the data description. The signature is used to match against the type of a variable, to determine whether the data description should be used for the variable. Wildcards can be used when matching. For example, to match an STL list, use the format:

`signature = {"std::list<*>"}`

To match objects or pointers of either type `DeviceType1` or `DeviceType2`, use the format:

`signature = {"DeviceType1", "DeviceType2"}`

Variables are always cast to derived types and dereferenced before being used in a data description.

This field is required.

`required_fields = {"field1", "field2", ... }`

Specifies a list of fields (*field1*, *field2*, etc.) that must be present in the type to match this data description. This line is not required, but specifying required fields can be useful for sanity checking.

```
type = "data_desc_type"
```

Specifies what type of data description is being defined, where *data_desc_type* is one of the following:

- container
- list
- null_terminated_list
- circular_list
- binary_tree
- structure
- alias
- singleton
- function_definer
- graph

See “Data Description Types and Their Type-Specific Fields” on page 581 for more information about each data type.

This field is required.

```
predicate = "pred"
```

Specifies a predicate expression to determine whether the data description should be used. For example, if this data description should only be used when the *value* field of the struct is 0, the predicate line of the description would be:

```
predicate = "return self.value == 0"
```

Specifying a predicate is optional.

required and optional fields depending on type

Specifies required and optional fields according to the type of data description. See the following sections for a description of the required and optional fields for each data type.

When checking whether a data description matches the variable, MULTI first casts the variable to its derived type. Then MULTI compares the type name to the *signature* entries in the data description. If the type name matches, MULTI checks that all fields listed in *required_fields* are present in the structure, and then evaluates the *predicate* expression, if any. If the

type matches the signature, `required_fields`, and the predicate expression returns `true`, the data description is used to display that type.

Data Description Types and Their Type-Specific Fields

The available data types are described in the sections below, along with the required and optional data description fields for each.

The container Data Type

The `container` type is used to describe a type which is a container of elements, such as a list, vector, or map. A data description for a container includes information about how to create an *iterator* for the container, how to walk the iterator through the container, and how to retrieve a value from the iterator at each step. An iterator is a value that represents a position in the container. For example, for a linked list, the iterator might be the address of a node.



Tip If extra information about the parent structure is needed by an iterator, set a MULTI special variable to contain this information in the `begin_iter` expression (see below).

An example of a container data description for an STL list is shown below.

```
list {
  signature = {"list<*>", "std::list<*>"}
  required_fields = {"_Mysize", "_Myhead"}
  type = "container"

  size = "return self._Mysize;"
  begin_iter = "return self._Myhead._Next;"
  next_iter = "return self._Next;"
  value_from_iter = "return self._Myval;"
}
```

The type-specific fields for the `container` data description type are described in the following table.

<pre>size = "size_expr"</pre> Specifies an expression, <i>size_expr</i>, that returns an integer value that equals the number of elements in the container. This field is required.
<pre>begin_iter = "begin_iter_expr"</pre> Specifies an expression, <i>begin_iter_expr</i>, that returns some representation of a beginning iterator over the container. For example, a list data description might return a pointer to the first list node. A vector might return an index to the first element. This field is required.
<pre>next_iter = "next_iter_expr"</pre> Specifies an expression, <i>next_iter_expr</i>, that is used to retrieve the next iterator from the current iterator. In this expression, the <code>self</code> variable refers to the current iterator, not to the original structure. For example, for a list, this line might be: <pre>next_iter = "return self.next"</pre> This field is required.


```
value_from_iter = "value_from_iter_expr"
```

Specifies an expression, *value_from_iter_expr*, that is used to retrieve the element value from the current iterator. In this expression, the `self` variable refers to the current iterator, not the original structure.

For example, for a list, this line might be:

```
value_from_iter = "return self.value"
```

This field is required.

```
use_for_print_and_view = bool_expr
```

Specifies whether the data description should be used for displaying the type in Data Explorers and when printing, where *bool_expr* is either `true` or `false`.

Defining this field is optional. It defaults to `true`, which causes the graph description to be used for graph views and print views. In most situations, this setting should not be changed; however, there may be times when you need to set up a container data description to support another data description (particularly in graph data descriptions), but do not want the container to display in a Data Explorer.

The list Data Type

The `list` type is a specialized type of container used to describe C-style lists. The list is made up of structures that are both list nodes and elements, each connected to its successor by a *next* pointer. This type of data description is useful for lists that rely on a termination condition, rather than a size parameter.

For example, consider code using the following structure definition:

```
struct ListType {  
    int value;  
    struct ListType *next;  
};  
struct ListType EndOfListNode;
```

To view a `ListType` variable as a list instead of simply a structure with a `next` pointer, you could use the following data description:

```
ListTypeDataDescription {  
    signature = {"ListType"}  
    required_fields = {"value", "next"}  
    type = "list"  
  
    next = "next"  
    termination_condition = "return &self == &EndOfListNode"  
}
```

The type-specific fields for data descriptions for lists are described in the table below.

```
next = "next_field_name"
```

Specifies *next_field_name* as the field within the type that points to the next element in the list.

This line is required.

```
termination_condition = "term_condition_expr"
```

Specifies the expression *term_condition_expr* as the condition that ends the list.

This field is required.

```
use_for_print_and_view = bool_expr
```

Specifies whether the data description should be used for displaying the type in Data Explorers and when printing, where *bool_expr* is either `true` or `false`.

Defining this field is optional. It defaults to `true`, which causes the graph description to be used for graph views and print views. In most situations, this setting should not be changed; however, there may be times when you need to set up a container data description to support another data description (particularly in `graph` data descriptions), but do not want the container to display in a Data Explorer.

The null_terminated_list Data Type

The `null_terminated_list` type is a specialized type of list used to describe standard C-style null-terminated lists.

For example, consider code using the following structure definition:

```
struct NullTerminatedListType {
    int value;
    struct NullTerminatedListType *next;
};
```

To view a `NullTerminatedListType` variable as a list instead of simply a structure with a `next` pointer, you could use the following data description:

```
NullTerminatedListTypeDataDescription {
    signature = {"NullTerminatedListType"}
    required_fields = {"value", "next"}
    type = "null_terminated_list"

    next = "next"
}
```

The type-specific fields for data descriptions for null-terminated lists are described in the table below.

<pre>next = "next_field_name"</pre>
Specifies <i>next_field_name</i> as the field within the type that points to the next element in the list. This line is required.
<pre>use_for_print_and_view = bool_expr</pre> Specifies whether the data description should be used for displaying the type in Data Explorers and when printing, where <i>bool_expr</i> is either <code>true</code> or <code>false</code>. Defining this field is optional. It defaults to <code>true</code>, which causes the graph description to be used for graph views and print views. In most situations, this setting should not be changed; however, there may be times when you need to set up a container data description to support another data description (particularly in <code>graph</code> data descriptions), but do not want the container to display in a Data Explorer.

The circular_list Data Type

The `circular_list` type is a specialized type of list used to describe C-style circular lists (i.e., lists in which the end is discovered by returning to an element that has already been seen).

For example, consider code using the following structure definition:

```
struct CircularListType {
    int value;
    struct ListType *next;
};
```

To view a `CircularListType` variable as a list instead of simply a structure with a `next` pointer, you could use the following data description:

```
CircularListTypeDataDescription {
    signature = {"CircularListType"}
    required_fields = {"value", "next"}
    type = "circular_list"

    next = "next"
}
```

The type-specific fields for data descriptions for circular lists are described in the table below.

```
next = "next_field_name"
```

Specifies *next_field_name* as the field within the type that points to the next element in the list.

This line is required.

```
use_for_print_and_view = bool_expr
```

Specifies whether the data description should be used for displaying the type in Data Explorers and when printing, where *bool_expr* is either `true` or `false`.

Defining this field is optional. It defaults to `true`, which causes the graph description to be used for graph views and print views. In most situations, this setting should not be changed; however, there may be times when you need to set up a container data description to support another data description (particularly in graph data descriptions), but do not want the container to display in a Data Explorer.

The `binary_tree` Data Type

The `binary_tree` type is used to describe a type of container used to describe a binary tree structure. The tree is traversed in order — first the `left` subtree is traversed, then the value of the node is displayed, then finally the `right` subtree is traversed.

For example, consider code using the following structure definition:

```
struct TreeNode {
    int value;
    struct TreeNode *left, *right;
}
```

To view a `TreeNode` variable as an in-order list of elements rather than simply as a structure, you could use the following data description:

```
TreeNodeDataDescription {
    signature = {"TreeNode"}
    required_fields = {"value", "left", "right"}
    type = "binary_tree"

    left = "left"
    right = "right"
}
```

The type-specific fields for data descriptions for binary trees are described in the following table.

```
left = "left_field_name"
```

Specifies *left_field_name* as the field representing the left child pointer.

This field is required.

```
right = "right_field_name"
```

Specifies *right_field_name* as the field representing the right child pointer.

This field is required.

```
use_for_print_and_view = bool_expr
```

Specifies whether the data description should be used for displaying the type in Data Explorers and when printing, where *bool_expr* is either `true` or `false`.

Defining this field is optional. It defaults to `true`, which causes the graph description to be used for graph views and print views. In most situations, this setting should not be changed; however, there may be times when you need to set up a container data description to support another data description (particularly in `graph data` descriptions), but do not want the container to display in a Data Explorer.

The structure Data Type

The `structure` type is used to describe a list of artificial fields that can be viewed for the data type. The data type is seen as a structure containing the fields defined in the `structure` data description.

The syntax for the `structure` type data description follows (lines enclosed in square brackets are optional; the square brackets are not part of the syntax, but the curly brackets are):

```
data_desc_name {  
  signature = {"sig1", "sig2", ... }  
  [required_fields = {"field1", "field2", ... }]  
  type = "structure"  
  [predicate = "pred"]
```

```
fields {
  field1{
    name = name_string
    value = "value_expr"
    mutable = bool_expr
  }
  ...
}
```

An example of a structure data description for an STL list iterator is shown below.

```
list_iterator {
  signature = {"list<*>::*iterator"}
  required_fields = {"_Ptr"}
  type = "structure"
  predicate = "return self._Ptr != 0"

  fields {
    next {
      name = "_Next"
      value = "return self._Ptr->_Next"
    }
    prev {
      name = "_Prev"
      value = "return self._Ptr->_Prev"
    }
    value {
      name = "_Myval"
      value = "return self._Ptr->_Myval"
    }
  }
}
```

The type-specific fields for data descriptions for structures are described in the table below.

<code>field1</code>
A unique identifier for the field description.
<code>name = "name_string"</code> Specifies <i>name_string</i> as the name of the artificial field you are creating. This name must not contain any white spaces. This field is required.
<code>value = "value_expr"</code> Specifies the expression, <i>value_expr</i> , as the value to be displayed for the field. This field is required.
<code>mutable = bool_expr</code> Specifies whether the value can be edited in Data Explorer windows, where <i>bool_expr</i> is either <code>true</code> or <code>false</code> . Defining this field is optional. It defaults to <code>true</code> , which allows the values to be edited.

The alias Data Type

The `alias` type is used to specify that one type should be viewed as if it were a different type.

An example of an `alias` data description for an STL string is shown next. This data description causes an STL string to be displayed as its underlying C-style string.


```
string {
    signature = {"basic_string<*>"}
    required_fields = {"_Bx", "_Myres"}
    type = "alias"

    value = "if (_BUF_SIZE <= self._Myres) {"
    value += "    return self._Bx._Ptr;"
    value += "} else {"
    value += "    return self._Bx._Buf;"
    value += "}"
    mutable = false;
}
```

The type-specific fields for data descriptions for aliases are described in the table below.

```
value = "val_expr"
```

Specifies that the result of the expression *val_expr* will be displayed in place of the actual value of the item matching this data description.

This field is required.

```
mutable = bool_expr
```

Specifies whether the value can be edited in Data Explorer windows, where *bool_expr* is either `true` or `false`.

Defining this field is optional. It defaults to `true`, which allows the value to be edited.

```
replace_self = bool_expr
```

Specifies whether the variable being viewed is replaced with the result of the expression specified by the line `value = "val_expr"`. The argument *bool_expr* is either `true` or `false`. If *bool_expr* is `true`, the variable being viewed is replaced with the result of *val_expr*. If *bool_expr* is `false`, the result of *val_expr* is used as the value of the variable, but the original variable's type is still displayed.

Defining this field is optional. It defaults to `false`.

The singleton Data Type

The singleton type is used to specify a data type that should be viewed as a singleton type. Since the data type is a singleton, only one instance of it is ever created. Specifying a data description for the singleton allows you to view the instance of the data type by viewing the type.

For example, a singleton data description for the following type:

```
class SingletonClass {
    static SingletonClass* getInstance () {
        static SingletonClass* instance = NULL;
        if (!instance)
            instance = new SingletonClass;
        return instance;
    }
}
```

would be:

```
singleton_class {
    signature = {"SingletonClass"}
    type = "singleton"

    instance = "SingletonClass::getInstance#instance"
}
```

This data description would allow you to see the singleton instance of the class by viewing the class type. There is only one type-specific field for data descriptions for singletons, which is described in the table below.

```
instance = "variable_name"
```

Specifies *variable_name* as the instance variable of the type. This must be a valid variable identifier.

This field is required.

The function_definer Data Type

The `function_definer` type is used to define macros that can be called from the Debugger command line as if they were member functions of the data type.

The syntax for the `function_definer` data description follows (lines enclosed in square brackets are optional; the square brackets are not part of the syntax, but the curly brackets are):

```
data_desc_name {
  signature = {"sig1", "sig2", ... }
  [required_fields = {"field1", "field2", ... }]
  type = "function_definer"
  [predicate = "pred"]

  functions {
    unique_name {
      name = "name_string"
      [arguments = {"arg_string1", "arg_string2", ...}]
      body = "body_expr"
    }
  }
}
```



Note The functions block shown above can also be added to any of the other data description types (except graph), rather than being placed in a `function_definer` type.

A sample `function_definer` data description for an STL list is shown next.

```
list_funcs {
  signature = {"list<*>"}
  required_fields = {"_Mysize", "_Myhead"}
  type = "function_definer"

  functions {
    size {
      name = "size"
      body = "return self._Mysize"
    }
    front {
      name = "front"
      body = "return self._Myhead->_Next->_Myval"
    }
    back {
      name = "back"
      body = "return self._Myhead->_Prev->_Myval"
    }
  }
}
```

The type-specific fields for data descriptions for function definers are described in the table below.

<code>unique_name</code>
A unique name for the function entry.
<code>name = "name_string"</code>
Specifies the string <i>name_string</i> as the name of the function. For example, if the name is <code>size</code> , then the function can be called by <code>obj.size()</code> . This field is required.
<code>arguments = {"arg_string1", "arg_string2", ...}</code>
Specifies additional arguments (<i>arg_string1</i> , <i>arg_string2</i> , ...) to the function. Like C++ class member functions, there is always one implicit argument to the function— <code>self</code> —which contains a pointer to the object being referenced (taking the place of the <code>this</code> parameter in C++). This line is optional.
<code>body = "body_expr"</code>
Specifies an expression, <i>body_expr</i> , that is evaluated when the function is called. This field is required.

The graph Data Type

The graph type is used to describe a data type that should be displayed graphically in a **Graph View** window. The data description defines how a data type will be represented in nodes in a graph, and contains information about the text that should appear in the nodes, the children of the nodes, and the parents of the nodes.

Consider the following type:

```
class Device {
public:
    const char* deviceName;
    int deviceId;

    Device(const char *name, const int id);
    void addInputBus(Bus* input);
    void addOutputBus(Bus* output);

protected:
    std::list<Bus*> inputs;
    std::list<Bus*> outputs;
};
```

This type might have the following data description:

```
device {
    signature = {"Device"}
    type = "graph"
    node_text = "mprintf(\"%s\\nDevice ID: %d\\n\", self.deviceName, self.deviceId)"
    children {
        outputs {
            value = "return self.outputs"
            is_container = true
        }
    }
    parents {
        inputs {
            value = "return self.inputs"
            is_container = true
        }
    }
}
```

See “.mdv File Examples” on page 605 for a complete example of the graph data description.

The type-specific fields for data descriptions for graphs are described in the table below.

```
replace_with = "repl_node_expr"
```

Specifies a value, *repl_node_expr*, to replace (in **Graph View**) the node being defined. If this line is present, all other fields in the data description are ignored. The node being defined will not show up in the graph, but will instead be replaced by a node representing the value returned by the expression *repl_node_expr*.

To make a node not show up at all, use the line:

```
replace_with = "return (void*) 0"
```

This field is optional.

```
node_text = "commands"
```

Specifies one or more MULTI commands, separated by semicolons, that generate output which will appear on the node being defined. Commands that can be used for output include **echo**, **print**, and **mprintf**. Any textual output from these commands will be captured and displayed in the node.

This field is required.

```
vertical_layout = bool_expr
```

Specifies whether the graph should use a vertically-oriented layout or a horizontally-oriented layout, where *bool_expr* is `true` or `false`. If *bool_expr* is `true`, the graph will use a vertically-oriented layout.

This field is optional and defaults to `false`. This field can be set on a profile-by-profile basis (see “Profile Descriptions” on page 600).

```
children {  
  child_name {  
    value = "child_val_expr"  
  }  
  ...  
}
```

Specifies information about the node’s children. Each child must have a unique name, *child_name*, but the number of children is unlimited.

For more information about the syntax and fields for describing children, see the next section.

Specifying children is optional.

```
parents {  
  parent_name {  
    value = "parent_val_expr"  
  }  
  ...  
}
```

Specifies information about the node's parents. Each parent must have a unique name, *parent_name*.

If an edge is specified by both the parent and the child, it will only appear once. This provides an easy mechanism for describing complex graphs without having to worry about duplication.

For more information about the syntax and fields for describing parents, see the next section.

Specifying parents is optional.

```
next_sibling = "sibl_expr"
```

Specifies information about the node's siblings, where *sibl_expr* is an expression.

Siblings are laid out orthogonally to the primary layout orientation. If the graph is horizontally oriented, siblings will be connected vertically. This provides a convenient way to display data in a list of trees. There can be at most one sibling edge entering a node, and one sibling edge leaving a node.

Specifying siblings is optional.

```
max_display_size = int
```

Specifies the maximum number, *int*, of siblings displayed in a chain.

This field is optional and defaults to 20, meaning that if you have a list being displayed as siblings, it will get cut off after the twentieth element.

Describing Parents and Children of the Node

The core functionality of the graph type is describing children and parents of a node. The children of a node may be significant pointers contained in the structure, entries from a lookup table, or any other value that can be calculated giving the structure as a starting point.

The syntax for defining a child follows (lines enclosed in square brackets are optional; the square brackets are not part of the syntax, but the curly brackets are):

```
children {  
  child_name {  
    value = "child_val_expr"  
    [is_container = bool_expr]  
    [is_array = bool_expr]  
    [array_len = "length_expr"]  
    [invisible_edge = bool_expr]  
  }  
  ...  
}
```

The lines in this syntax are described in the following table.


```
value = "child_val_expr"
```

Specifies the value of the child, where *child_val_expr* is an expression.

For example, in a binary tree you might define one child with:

```
value = "return self.left"
```

and another with:

```
value = "return self.right"
```

You must specify a value for every child.

```
is_container = bool_expr
```

Specifies whether this child should be treated as a container of children, where *bool_expr* is either `true` or `false`.

If *bool_expr* is `true`, the child is treated as a container — MULTI looks up a data description for that type and adds each element of the container as a child. STL container descriptions have been provided, so any time you have a structure with a list, vector, or map of children, you can simply set this value to `true` and each element will be added as a child.

This field is optional and defaults to `false`. This field is mutually exclusive with *is_array*; only one of *is_array* and *is_container* can be `true`.

```
is_array = bool_expr
```

Specifies whether the child should be treated as an array of children, where *bool_expr* is either `true` or `false`.

This field is optional and defaults to `false`. This field is mutually exclusive with *is_container*; only one of *is_array* and *is_container* can be `true`.

```
array_len = "length_expr"
```

Specifies an expression, *length_expr*, to be used to calculate the length of the array. An example child using arrays might be:

```
value = "return self.myChildArray"  
is_array = true  
array_len = "return self.myChildArrayLen"
```

This field is required if `is_array = true`.

```
invisible_edge = bool_expr
```

Specifies whether the edge defined by this parent or child relationship will be displayed, where *bool_expr* is either `true` or `false`.

This field gives you greater control over the layout of a graph. If *bool_expr* is `true`, the edge will not be displayed, but will still restrain the layout of the graph, allowing you to, for example, enforce an ordering on nodes without having visible edges connecting them.

This field is optional and defaults to `false`.

Parents can be described in the same way as children by using the `parents` keyword in place of the `children` keyword. A single data description can contain both `parents` and `children` blocks.

Profile Descriptions

A data visualization profile is a collection of data descriptions. You can create several profiles, but only one is *active* at a given time. Only the active profile is used when searching for data descriptions. Using profiles allows you to have multiple data descriptions for the same type, which you use in different circumstances. You can change which profile is active by using the command:

```
dvprofile prof_name
```

where *prof_name* is the name of the profile to be made active.

Any data description not in a profile is placed in a global profile, and is always accessible.

Profiles are defined in profile descriptions in **.mdv** files. A profile description has the following form. (Lines enclosed in square brackets are optional; the

square brackets are not part of the syntax, but the curly brackets are part of the syntax.)

```
unique_prof_id {
  profile_name = "prof_name"
  [parent_profile = "parent_prof_name"]
  [default_root = "def_root_prof"]
  [vertical_layout = bool_expr]
  [add_siblings_of_root = bool_expr]

  data_descriptions {
    [series_of_data_descriptions]
  }
}
```

The lines in this syntax are described in the table below.

```
unique_prof_id
```

Specifies a unique identifier, *unique_prof_id*, for the profile. Reading in a new profile with the same identifier will overwrite the original.

This field is required.

```
profile_name = "prof_name"
```

Specifies *prof_name* as the string used to refer to the profile when you are using the **dvprofile** command or working with different views. See the **dvprofile** command in “Data Visualization Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book, and see “View Descriptions” on page 602.

This field is required.

```
parent_profile = "parent_prof_name"
```

Specifies that the profile with the name *parent_prof_name* is the parent of the profile. A profile can have at most one parent. When searching for a data description, MULTI starts at the active profile and then searches its parent, then the parent of that parent, and so on. If no parent is specified, the parent is set to the global profile by default.

This field is optional.

```
default_root = def_root_prof
```

Specifies a variable identifier *def_root_prof* as the default root variable for graphs using this profile. This line is optional, but is used by the **dataview** command. For information about the **dataview** command, see “Data Visualization Commands” in Chapter 22, “View Command Reference” in the *MULTI: Debugging Command Reference* book.

```
vertical_layout = bool_expr
```

Specifies whether the graph should use a vertically oriented layout or a horizontally oriented layout, where *bool_expr* is either `true` or `false`. If *bool_expr* is `true`, the graph will use a vertically oriented layout.

This field is optional and defaults to `false`.

```
add_siblings_of_root = bool_expr
```

Specifies whether the siblings of the root node for the graph of this profile should be added, where *bool_expr* is either `true` or `false`. If *bool_expr* is `true`, siblings will be added.

This field is optional and defaults to `true`.

View Descriptions

Views are high-level visualizations in graph form. You can create views for elements of the graph data type by including *view descriptions* in your loaded **.mdv** file(s). When you select **Display Program Visualizations** from a right-click shortcut menu in the Debugger source pane, a graph will be displayed for each view that is defined in your loaded **.mdv** files. These individual graphs will appear on separate tabs of a single window.

Every view starts at a default root and contains one or two panes. A single pane view displays the data specified by a single profile. A dual-pane view displays a profile in the initial, or primary, pane. When a node in this primary pane is clicked, that node is used as the root variable for a graph that is then displayed in the second pane, using a different profile.

A view description contains specifications about the contents and layout of a single view. The syntax for defining views has the following form. (Lines enclosed in square brackets are optional; the square brackets are not part of the syntax, but the curly brackets are part of the syntax.)

```

view_unique_name {
  initial_pane = "pane_name"
  default_root = "def_root_view"
  [tab_name = "tab"]

  panes {
    pane_name1 {
      profile_name = "prof_name"
      [child_pane = "child_pane_name"]
    }
    [pane_name2 {
      profile_name = "prof_name2"
    }]
  }
}

```

The fields in this syntax are described in the table below.

view_unique_name

Specifies a unique identifier, *view_unique_name*, for the view. This name can be used with the **dataview** command to open a **Graph View** window on this view (see “Invoking Customized Data Visualizations” on page 577).

This field is required.

initial_pane = "pane_name"

Specifies which pane is the primary pane for this view. In single pane views, this will be the name of the only pane. In dual pane views, this will be the name of the pane on the left. In dual pane views, clicking a node in the primary pane will cause the child pane to display a new graph with the selected object as its root.

This field is required.

default_root = "def_root_view"

Specifies *def_root_view* as the root variable for the view.

This field is required.

tab_name = "tab"

Specifies the string to be displayed on the tab that displays this view.

This field is optional.

pane_name

Specifies the name (*pane_name1*, *pane_name2*) of the pane(s). This is used to refer to the pane within this view.

This field is required for each pane in the view.

profile_name = "*prof_name*"

Specifies *prof_name* as the profile to use as the active profile when displaying this pane. (See "Profile Descriptions" on page 600.)

This field is required for each pane in the view.

child_pane = "*child_pane_name*"

Specifies *child_pane_name* as the child pane. When a node is selected in the pane being defined, the selected node will be displayed in this child pane.

This field is required if the pane has a child pane.

Using Expressions in MULTI Data Visualization (.mdv) Files

Many of the fields defined in **.mdv** files take an expression as their value. An expression is specified in quotation marks and can contain any statement or series of statements that can be evaluated from the MULTI command pane. The value of the expression is whatever value is found in the `return` statement. If no `return` statement is specified, the expression returns `void` and is probably incorrect.

Expressions can contain loops, MULTI special variables, and command line procedure calls. Expressions can also refer to program variables and return pointers, but they cannot return objects.

Expressions access the *current* structure through the `self` keyword. The current structure is usually the structure that the data description's signature matches. (There are some exceptions, such as when the current structure is the iterator while walking through a container.)

.mdv File Examples

Consider the following data types:

```
class Device {
public:
    Device(const char *name, const int id);
    void addInputBus(Bus* input);
    void addOutputBus(Bus* output);

protected:
    const char* deviceName;
    int deviceId;

    std::list<Bus*> inputs;
    std::list<Bus*> outputs;
};

class Bus {
public:
    Bus(const char* name, const int id);
    void addInputDevice(Device* input);
    void addOutputDevice(Device* output);

protected:
    const char* busName;
    int busId;

    std::list<Device*> inputs;
    std::list<Device*> outputs;
};
```

A sample **.mdv** file used to display the preceding types is listed below.

```
demo_profile {
  profile_name = "Device_Bus_Profile"

  data_descriptions {
    device {
      signature = {"Device"}
      type = "graph"
      node_text = "mprintf(\"%s\nDevice ID: %d\", self.deviceName, self.deviceId)"
      children {
        outputs {
          value = "return self.outputs"
          is_container = true;
        }
      }
      parents {
        inputs {
          value = "return self.inputs"
          is_container = true;
        }
      }
    }
  }

  bus {
    signature = {"Bus"}
    type = "graph"
    node_text = "mprintf(\"%s\nBus ID: %d\nInputs: %d\nOutputs: %d\", "
    node_text+= "self.busName, self.busId, self.inputs.size(), self.outputs.size())"
    children {
      outputs {
        value = "return self.outputs"
        is_container = true;
      }
    }
    parents {
      inputs {
        value = "return self.inputs"
        is_container = true;
      }
    }
  }
}
```



```

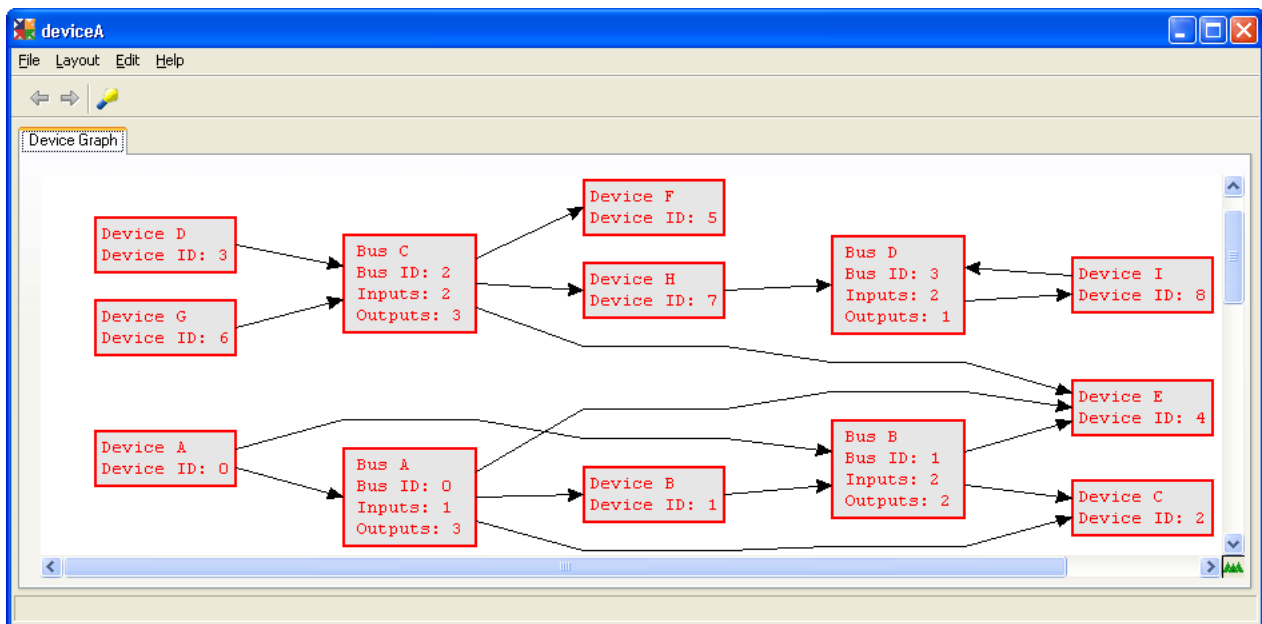
my_list {
    signature = {"std::list<*>"}
    type = "container"

    size = "return self._Mysize;"
    begin_iter = "return self._Myhead._Next;"
    next_iter = "return self._Next;"
    value_from_iter = "return self._Myval;"
}
}

demo_view {
    initial_pane = "only_pane"
    default_root = "deviceA"
    tab_name = "Device Graph"
    panes {
        only_pane {
            profile_name = "Device_Bus_Profile"
        }
    }
}

```

The preceding .mdv file might yield a **Graph View** like the following.



Register Definition and Configuration Reference

Appendix E

This Appendix Contains:

- The GRD Register Definition Format
- Register View Window Configuration Files

This appendix describes the GRD register definition format, and provides a description of **Register View** window Configuration Files.



Note The legacy RDF format is no longer supported; it has been superseded by the GRD format.

The GRD Register Definition Format

A target's register information is defined in a GRD file, which will contain some or all of the following sections:

- The `general` section
- The `enum` section
- The `structure` section
- The `bitfield` section
- The `register` section
- The `group` section

Each section is enclosed in a block (delimited by `{ }`) with a tag name identifying the section. The definition for each section can be separated into multiple parts. For example, you can define one `register` section for all special purpose registers, then a `group` section for the special purpose registers, and then another `register` section.

Preprocessor directives can be used in GRD files. The syntax and usage of the GRD file preprocessor is the same as in C++ except that the initial character is `%` instead of `#` (an initial `#` indicates a comment line in GRD format).

A preprocessor symbol can be substituted in a string with syntax `${DefinedSymbol}`. For example:

```
%define PPC400_DCR_BASE    5000

CPC0_SR {address="${PPC400_DCR_BASE}+0x000000b0"}
```

When the Debugger loads them, the string for address of `CPC0_SR` will be replaced into `5000+0x000000b0`, which will then be evaluated.

Blocks can be represented in two formats:

- Delimited by curly braces ({ }). For example:

```
general {  
    register_offset = "16";  
}
```

- Delimited with a period (.). For example:

```
general.register_offset = "16";
```

or

```
"general"."register_offset" = "16";
```

The following sections describe each element's syntax.

The general Section

The general section defines the following settings for the target:

```
general {  
    version = number  
    pad_bitfield = bool  
    register_width = int  
    register_offset = string  
    byte_endian = string  
    bit_0_is_msb = bool  
    memory_space = number  
    pvr_address[index] = identifier  
    pvr_info[index] {register definition block}  
    pvr_mask[index] = mask  
    bcr_address[index] = identifier  
    bcr_info[index] {register definition block}  
    bcr_mask[index] = mask  
    multi_command = commands  
}
```



Note The only required attribute is `version`. All other attributes are optional.

`version = number`

Defines the version number of the Green Hills Software Register Definition language used in the GRD file. The version number should be 2.

Example: `version = 2`

`pad_bitfield = bool`

Where *bool* can be:

- `true` or `false`
- An integer (where 0 means false, and a non-zero integer means true)
- A string expression starting with `%EVAL` that can be dynamically evaluated and result in an Boolean value

When a register with bitfields is defined, some bits or bit ranges may be omitted. This option tells the Debugger whether to automatically generate fields for the omitted bits or bit ranges when creating symbol information for the bitfield. If this attribute is true, when a register with the corresponding bitfield is displayed, the automatically generated padding fields will be displayed.

The default value for this attribute is `true`.

Example: `pad_bitfield = true`

`register_width = int`

Where *number_of_bits* defines the default register width in bits. It can be an integer or a string expression starting with `%EVAL`. The expression enclosed in `%EVAL { }`, will be dynamically evaluated into an integer.

The default value for this attribute is 32.

The default register width will be applied to all registers except those registers whose definitions override the default width by using the `width` setting.

Example: `register_width = 16`

`register_offset = string`

Each register has an identifier in the Debugger, but it can have a different identifier (register offset) in the communication message between the Debugger and the debug server. This setting defines the default register offset that will be applied to all registers. A register definition can override the default offset expression by using the `offset` setting.

Example: `register_offset = "return 2*__regadr"`

`__regadr` is a reserved symbol whose value is the register's identifier in the Debugger.

```
byte_endian = string
```

This setting defines the default byte order for a register. A register definition can override the default byte order by using the `byte_endian` setting.

Here are the supported values:

- `default` — Always the same as the target (this is the default)
- `big` — Always big endian
- `little` — Always little endian

In big endian registers, the most significant byte is at the lowest address. In little endian registers, the least significant byte is at the lowest address.

Example: `byte_endian = "big"`

```
bit_0_is_msb = bool
```

This setting defines the default bit order for a register. A register definition can override the default bit order by using the `bit_0_is_msb` setting.

This setting indicates how a register's bit numbers are ordered. If the value for `bit_0_is_msb` is true, bit 0 will refer to the register's the most-significant bit (msb) and the largest bit number will refer to the register's least-significant-bit (lsb). Otherwise, bit 0 will refer to the register's least-significant bit (lsb) and the largest bit number will refer to the register's most-significant-bit (msb)

The default value for the attribute is `false`.

Example: `bit_0_is_msb = true`

```
memory_space = number
```

This setting defines the default memory space for memory-mapped registers.

When defining a custom memory space, you should use a value larger than 64 KB. The Debugger reserves values less than 64 KB for its own use.

The Debugger also supports the following memory spaces, which are defined in the **os_constants.grd** file located at *install_dir/defaults/registers*.

- `MSPACE_TEXT_PHYSICAL` — The physical memory space for text.
- `MSPACE_DATA_PHYSICAL` — [default] The physical memory space for data.
- `MSPACE_TEXT_DEFAULT` — The default memory space (recognized by the debug server) for text.
- `MSPACE_DATA_DEFAULT` — The default memory space (recognized by the debug server) for data.

See also the `memory_space` attribute in "The register Section" on page 626.

Example: `memory_space = 95`

```
pvr_address[index] = identifier
```

If a processor has more than one processor version register (PVR), *index* can be appended to *pvr_address*. The two settings *pvr_address* and *pvr_address0* represent the same processor version register.

identifier can be an integer or a string expression starting with %EVAL that will evaluate to an integer.

See also “Processor Version Register Information” on page 617.

Example: *pvr_address10 = 1061*

```
pvr_info[index] {register definition block}
```

This setting defines a memory-mapped processor version register (PVR). To support memory-mapped PVRs, the syntax for the PVR definition is extended to support attributes from the register definition block. For information about the register definition block, see “The register Section” on page 626.

The *access* string must be a regular, special, io, or memory-mapped string, or it must be a string expression that can be resolved into such a string. For more information, see the *access* attribute in “The register Section” on page 626.

See also “Processor Version Register Information” on page 617.

When this PVR specification is used, the version number should be 2. See the *version* description earlier in this table.

Example:

```
pvr_info1 {  
    access = "memorymapped";  
    address = 0x123456;  
    width = 16;  
    read_length = 16;  
    byte_endian="little";  
}
```

```
pvr_mask[index] = mask
```

index has the same meaning as in the processor version register address.

mask can be an integer or a string expression starting with %EVAL that will evaluate to an integer.

See also “Processor Version Register Information” on page 617.

Example: *pvr_mask10 = 0xffff0000*


```
bcr_address[index] = identifier
```

If a target has more than one board configuration register (BCR), *index* can be appended to *bcr_address*. The two settings *bcr_address* and *bcr_address0* represent the same BCR.

See also “Board Configuration Register Information” on page 617.

Example: *bcr_address10* = 258

```
bcr_info[index] {register definition block}
```

This setting defines a memory-mapped board configuration register (BCR). To support memory-mapped BCRs, the syntax for the BCR definition is extended to support attributes from the register definition block. For information about the register definition block, see “The register Section” on page 626.

The *access* string must be a regular, special, io, or memory-mapped string, or it must be an expression that can be resolved into such a string. For more information, see the *access* attribute in “The register Section” on page 626.

See also “Board Configuration Register Information” on page 617.

When this BCR specification is used, the version number should be 2. See the *version* description earlier in this table.

Example:

```
bcr_info1 {
    access = "memorymapped";
    address = 0x123456;
    width = 16;
    read_length = 16;
    byte_endian="little";
}
```

```
bcr_mask[index] = mask
```

index has the same meaning as in the board configuration register address.

mask can be an integer or a string expression starting with %EVAL that will evaluate to an integer.

See also “Board Configuration Register Information” on page 617.

Example: `bcr_mask10 = 0xffff0000`

```
multi_command = multi_commands
```

```
multi_command += multi_commands
```

This setting defines Debugger commands in a string that will be executed when the register definition is constructed inside the Debugger.

This setting is especially useful to support memory-mapped registers whose addresses are not based on another register’s value but which could depend on other factors, such as the following:

- A debug server could move a register base around when it connects to the target (via its setup script, for example).
- A program such as the INTEGRITY kernel could move a register base when it runs.
- A program’s build process might move a register base.

You can define the address of these registers with an expression using one or more Debugger local variables. For example, on a PowerPC target supporting QUICC, a Debugger variable is defined for the base of the QUICC registers:

```
$MULTI_PPC_QUICC_SIU_IMMR_BASE
```

which can be used to define the addresses of QUICC registers as described below:

```
bcr_address = "$MULTI_PPC_QUICC_SIU_IMMR_BASE + 0x10024"
```

The special variable can be initialized to the default base with a Debugger command:

```
multi_command +=  
    " if (!$MULTI_PPC_QUICC_SIU_IMMR_BASE_IS_SET)  
{eval $MULTI_PPC_QUICC_SIU_IMMR_BASE = 0x0f000000;}"
```

where `eval` is used to prevent the Debugger from printing the new value of `$MULTI_PPC_QUICC_SIU_IMMR_BASE` after the assignment.

The default GRD files for some targets already use this mechanism. If the default IMMR base does not match your target setting, you can either change the GRD file directly or add statements such as the following into your program resource file (`.rc`) or target setup script (`.mbs`) to set the correct IMMR base and disable the default setting:

```
eval $MULTI_PPC_QUICC_SIU_IMMR_BASE_IS_SET = 1  
eval $MULTI_PPC_QUICC_SIU_IMMR_BASE = 0xfc000000
```

Processor Version Register Information

Some targets have a processor version register or registers that can be used to identify target processors and their variants. If such information about the processor version is defined, the value can be used in GRD file preprocessor expressions with the following syntax:

```
pvr_value[index]
```

where *index* is the index (starting from 0) for the processor version register. A processor version register's basic information contains:

- An identifier, which is used by the Debugger to communicate with the underlying debug server
- A value mask

Board Configuration Register Information

As with processor version registers, a mechanism is provided for users to define board configuration register(s). If such information for the board configuration registers is defined, the value can be used in GRD file preprocessor expressions with the following syntax:

```
bcr_value[index]
```

where *index* is the index (starting from 0) for the board configuration register.

A board configuration register's basic information contains:

- An identifier, which is used by the Debugger to communicate with the underlying debug server
- A value mask

The enum Section

An enumeration definition must be in an enumeration section, and must be defined before it is referenced. An enumeration can have the following attributes:

```
enum {  
    enum_name {  
        description = string  
        help_key = string  
        auto_value_desc = bool  
  
        enum_entry_name {  
            description = string  
            help_key = string  
            long_name = string  
            value = int  
        }  
        ...  
    }  
}
```



Note All attributes are optional.

```
description = string
description += string
desc = string
desc += string
```

The description string can be an expression starting with %EVAL, which will be dynamically evaluated.

An enumeration can have a long description combined from multiple description settings (with +=). Newline characters (\n) must be specified if you want a newline displayed in the Register Information window's help pane.

Example:

```
desc = "Indicates whether interrupts are enabled:\n";
desc += "    0 - disabled\n";
```

```
help_key = string
hk = string
```

The help key string can be an expression starting with %EVAL, which will be dynamically evaluated.

Example:

```
hk = "register.msr"
```

```
auto_value_desc = bool
```

The default setting for this option is `true`.

When this option is set to `false`, the Debugger will not automatically generate the specification for the enumeration's values in the help pane of the Register Information window for those bitfields whose type is the enumeration.

If you include a detailed list of enumeration values in the enumeration description, set this option to `false` to avoid duplicating the list of enumeration values in the help pane.

Example:

```
auto_value_desc = false
```

An enumeration must have at least one value entry, which can have the following attributes. If none of the attributes are specified, the braces should still be present.

```
description = string
description += string
desc = string
desc += string
```

The description string can be an expression starting with %EVAL, which will be dynamically evaluated.

```
help_key = string
hk = string
```

The help key will be used to show online help. The help key string can be an expression starting with %EVAL, which will be dynamically evaluated.

```
long_name = string
ln = string
```

The long name string can be an expression starting with %EVAL, which will be dynamically evaluated.

```
value = int
value = {int1, ..., intn}
```

The enumeration entry will have the value or values specified. If an enumeration entry does not have a value explicitly specified, its value is assigned as follows:

- If no previous enumeration entry exists, the enumeration entry will be assigned the value 0.
- If a previous enumeration entry exists, the enumeration entry will be assigned a value one greater than that of that previous enumeration entry.
- If a previous enumeration entry contains a list of values, the enumeration entry will be assigned a value one greater than maximum value in the previous enumeration entry.

The structure Section

Any structure definition must be in a structure section, and it must be defined before it is referenced. Structure definitions are rarely used in register definition files. A structure can have the following components:

```
structure {
    structure_name {
        description = string
        help_key = string

        field_name {
            description = string
            help_key = string
            long_name = string
            type = string
        }
        ...
    }
}
```



Note All attributes are optional.

`description = string`

`desc = string`

This is the structure's description. MULTI 5 does not use this value.

`help_key = string`

`hk = string`

This is the structure's help key. MULTI 5 does not use this value.

A structure's field can have the following attributes:

`description = string`

`desc = string`

This is the structure field's description. MULTI 5 does not use this value.

`help_key = string`

`hk = string`

This is the structure field's help key. MULTI 5 does not use this value.

```
long_name = string
```

```
ln = string
```

This is the structure field's long name. MULTI 5 does not use this value.

```
type = string
```

The type string must be one of the following:

- A basic data type, including
 - `int`
 - `unsigned int`
 - `char`
 - `unsigned char`
 - `short`
 - `unsigned short`
 - `long`
 - `unsigned long`
 - `long long`
 - `unsigned long long`
 - `float`
 - `double`
 - `code`
- A pointer to one of the above basic data types (for example, `int *`)

The bitfield Section

Any bitfield definition must be in a bitfield section, and must be defined before it is referenced.

A bitfield can have the following attributes:

```
bitfield {
    bitfield_name {
        description = string
        help_key = string

        field_name {
            description = string
            help_key = string
            short_name = string
            long_name = string
            auto_value_desc = bool
            loc = "begin_bit...end_bit"
            type = string
        }
        ...
    }
}
```



Note The only required attribute is `loc`. All other attributes are optional.

```
description = string
```

```
desc = string
```

The bitfield description is displayed in the Register Information window's help pane.

```
help_key = string
```

```
hk = string
```

This is the bitfield's help key. MULTI 5 does not use this value.

Every field in a bitfield must have a unique *field_name*.

```
description = string
description += string
desc = string
desc += string
```

A long description can be defined with multiple += statements.

The field description is displayed in the Register Information window's help pane.

```
help_key = string
hk = string
```

This is the field's help key. MULTI 5 does not use this value.

```
short_name = string
sn = string
```

This is the field's GUI name shown in **Register View** window and Register Information window. Multiple fields can have the same GUI name, like `Reserved` for those reserved fields. But each field's tag name must be unique in a bitfield.

If the attribute is absent, the field's tag name will be used as GUI name.

```
long_name = string
ln = string
```

This is the field's long name. It is displayed in the following places in the Register Information window:

- A tooltip in the concise display pane
- The **Description** in the detailed display pane
- The help pane

```
auto_value_desc = bool
```

This option is only relevant for fields with an enumeration type. The default setting for this option is `true`.

When this option is set to `false`, the Debugger will not automatically generate the specification for the field's values in the help pane of the Register Information window.

If you include a detailed list of enumeration values in the field description, set this option to `false` to avoid duplicating the list of enumeration values in the help pane.

```
loc = "begin_bit..end_bit"
```

```
loc = "bit_index"
```

The second form of this setting is for a field with only one bit.

```
type = type_string
```

The type must be one of the following basic types:

- hex
- binary
- unsigned
- signed
- A defined enumeration type

The register Section

A register definition must be in a register section. A register definition can have the following elements:

```
register {  
    register_name {  
        description = string  
        help_key = string  
        short_name = string  
        alias = {string_list}  
        long_name = string  
        gui_tab = string  
        cache_value = bool  
        access = string  
        address = number_or_string  
        offset = string  
        address_port = string  
        address_mask = number_or_string  
        data_port = string  
        width = number_or_string  
        byte_endian = string  
        bit_0_is_msb = bool  
        permission = string  
        type = string  
        read_length = number_or_string  
        write_length = number_or_string  
        hide = bool_or_string  
        memory_space = number  
    }  
}
```



Note The `address` attribute is always required. Other attributes may be required depending on the access type.

In the Debugger, each register has an identifier, which is usually defined by the `address` attribute. When the Debugger communicates with a debug server, a different register identifier may be used to refer to the register, in which case, the identifier is defined by the `offset` setting.

Some register attributes only apply to certain register types, as described in the following table.

```
description = string
description += string
desc = string
desc += string
```

This is the register's description. It is displayed in Register Information window.

```
help_key = string
hk = string
```

This is the register's help key. MULTI 5 does not use this value.

```
short_name = string
sn = string
```

If a register's short name is defined, the short name instead of the tag name will be displayed in Register Explorer windows.

```
alias = {string_list}
```

A register's aliases. They will be displayed in the **Register View** window, the Register Information window, and the Register Search window.

```
long_name = string
ln = string
```

This is the register's long name. It is displayed in the following places:

- As a tooltip in the **Register View** window
- In the help pane of the Register Information window

```
gui_tab = string
```

This setting allows a register to be associated with a **Register View** window tab. The Debugger will create the specified tab and display the register in the tab.

```
cache_value = bool
```

This setting indicates whether the Debugger can cache a register's value to improve performance while the target is halted.

The default value is `true`, which is okay for most registers. For some special registers, such as the timer, the setting should be defined as `false` so that when you manually refresh values from the **Register View** window or print values in the Debugger window, the values can be fetched from the target even though the corresponding process is stopped on the target.

`access = string`

The string must be one of the following strings or an expression that can be resolved into one of the following strings:

- `regular` — [default] A physical register that is accessed through the debug server. The address syntax is `num|expr`, which resolves to the register number.
- `fpv` — A floating point register that is accessed through the debug server. The address syntax is `num|expr`, which resolves to the register number.
- `special` — A special coprocessor register that is accessed through the debug server. The address syntax is `num|expr`, which resolves to the register number.
- `io` — An input/output register that is accessed through the debug server. The address syntax is `num|expr`, which resolves to the register number.
- `synonym` — A synonym for another defined register. The address syntax is `name|expr` and resolves to the name of the register for which this register is a synonym.
- `memorymapped` — The contents of the register correspond to a location in memory. The address syntax is `num|name|expr`.

If the address field is a number, it indicates the address in memory of the start of the register's contents.

If the address field is not a number, then it is assumed to be string containing a Debugger expression that will evaluate to an address. This expression is evaluated dynamically each time the register is accessed to produce an address, and may depend on other registers. This expression is *not* surrounded by `%EVAL{ }`.

An expression value surrounded by `%EVAL{ }` is evaluated when the file is loaded, producing either a numerical address or a string that contains a Debugger command to be evaluated at run time to produce an address.

- `dynamic` — The contents of the register are obtained by evaluating a Debugger expression. The address syntax is `name|expr`.

The value is a string that contains an expression in the language being debugged and which is evaluated to produce the register's value each time the register is accessed. This expression is *not* surrounded by `%EVAL{ }`.

An expression value surrounded by `%EVAL{ }` is evaluated when the file is loaded, producing a string that contains a Debugger command to be evaluated at run time to produce a value.

- `indirect` — The register's value is obtained by writing its address into an address port and reading its value from a data port. The address syntax is `num|expr` and resolves to the control value to be written to the address port so the register value will be made available on the data port.

`address = number_or_string`

For a register of type `regular`, `special`, `io`, or `fpu`, the address must be a number, or a string that can be immediately resolved into a number.

For a `synonym` register, the address should be another register name or a string that can be immediately resolved into another register name. The register name used here should not contain the `$` prefix.

For a `memorymapped` register, the address must be a number, or an expression that can be resolved into an address immediately or at the time it is accessed.

For a `dynamic` register, the address should be an expression that can be resolved into a number immediately or at the time it is accessed.

For an `indirect` register, the address can be a number or an expression that can be resolved into a number immediately or at the time it is accessed.

`offset = string`

The value must be a string or an expression that can be resolved into a number when the register is accessed.

The resolved number is the register's ID used by the Debugger to communicate with the debug server.

If the setting is absent and `register_offset` is defined in `General` section, then the string value of `register_offset` from `General` section will be used as the register's offset.

`address_port = number_or_string`

This setting defines an indirect register's address port.

The address port value must be a number, a register name or an expression that can be resolved into a number or register name. If it is an expression that can be resolved into a register name, it should be resolvable immediately.

`address_mask = number_or_string`

This setting defines the mask for the control value to be written to an indirect register's address port.

Its value must be a number or an expression that can be resolved into a number immediately.

`data_port = number_or_string`

This setting defines an indirect register's data port, from which the indirect register's value is read.

The data port value must be a number, a register name or an expression that can be resolved into a register name or a number. If it is an expression that can be resolved into a register name, it should be resolvable immediately.

An indirect register's data port and address port must both be register names or both be addresses.

`width = number_or_string`

This setting specifies a register's width in bits. If the attribute is not explicitly specified, the default register width will be used.

The value must be a number or a string expression that can be immediately resolved into a number.

`byte_endian = string`

This setting defines a register's byte order, and overrides the byte order defined in the `general` section. For the available values and other details, see `byte_endian` attribute in "The general Section" on page 611.

If the setting is absent, the byte order defined in the `general` section will be used for the register.

`bit_0_is_msb = bool`

This setting indicates how the register's bits are numbered in the bitfield definition, where:

- Bit 0 is the most-significant-bit (`msb`), or
- Bit 0 is the least-significant-bit (`lsb`).

If the attribute is not explicitly specified, the default bit order defined in the `general` section will be applied.

`permission = string`

The value must be a string in the syntax specified below or an expression which can be resolved into such a string immediately:

`action/permission[/user_mode] [;action/permission[/user_mode]]`

where:

- *action* can be one of the following values:
 - read
 - write
 - both [default]
- *permission* can be one of the following values:
 - none
 - once
 - full [default]
- *user_mode* can be one of the following values:
 - user
 - supervisor
 - both [default]

`type = string`

The value must be one of the following strings or an expression that can be immediately resolved into one of the strings:

- unsigned
- signed
- A defined bitfield name
- A defined enumeration name
- A defined structure name
- A pointer to one of the above types or a pointer to `void`, such as `unsigned *` or `void *`

`read_length = number_or_string`

The value must be a number or an expression that can be resolved into a number immediately.

The attribute is only valid for `memorymapped` registers or `indirect` registers located in memory. If no read length is specified for such registers, the default read block length is the length of the whole register.

`write_length = number_or_string`

The value must be a number or an expression that can be resolved into a number immediately.

The attribute is only valid for `memorymapped` registers or `indirect` registers located in memory. If no write length is specified for such registers, the default write block length is the length of the whole register.

`hide = bool_or_string`

The value must be a Boolean value or a string expression that can be resolved into a Boolean immediately.

This setting indicates whether a register should be hidden or displayed.

`memory_space = number`

This setting defines the memory space for memory-mapped registers.

For more information, see the `memory_space` attribute in “The general Section” on page 611.

The group Section

Registers appear in groups. A register must be included in at least one group, but a register can appear in multiple groups, if desired. If a register is not explicitly included by any group, it will be included in a default group named `Ungrouped`.

A group can contain other groups as well as registers. A group can be included by at most one other group. If a group is not included by any other group, it is a top-level group. A group must contain at least one register or group; otherwise, it will be automatically hidden.

Groups and registers compose the hierarchy in the Debugger’s **Register View** window. By default, only top-level groups are expanded and all nested groups are collapsed. A group may explicitly override the default collapse behavior.

A group definition can have the following elements:

```
group {
    group_name {
        description = string
        help_key = string
        short_name = string
        long_name = string
        top_level_index = number_or_string
        collapse = bool_or_string
        register = {registers}
        group = {groups}
    }
    ...
}
```



Note All attributes are optional.

```
description = string
description += string
desc = string
desc += string
```

This is the group's description. MULTI 5 does not use this value.

```
help_key = string
hk = string
```

This is the group's help key. MULTI 5 does not use this value.

```
short_name = string
sn = string
```

The value must be a string or an expression that can be immediately resolved into a string.

If group's short name is defined, it will be used to represent the group in the Debugger's **Register View** window, otherwise, the group's tag name is used.

```
long_name = string
ln = string
```

The value must be a string or an expression that can be immediately resolved into a string.

A group's long name is displayed in a tooltip in the **Register View** window.

```
top_level_index = number_or_string
```

The value must be a number or a string expression that can be immediately resolved into a number.

The attribute will be used to determine the group's position at top level if it is a top-level group, or inside a group if it is included by another group.

```
collapse = bool_or_string
```

The value must be a Boolean or a string expression that can be immediately resolved into a Boolean. The default value for this attribute is `true`.

If this attribute is `false`, the Debugger will expand the group when displaying it inside another group.

```
register = {registers}
```

```
register += {registers}
```

The strings used in the value to represent registers must be register tag names. The registers listed in the value must be defined.

```
group = {groups}
```

```
group += {groups}
```

The strings used in the value to represent groups must be group tag names. The groups listed in the value must be defined.

Register View Window Configuration Files



Note We suggest you avoid manually changing or creating **Register View** window configuration files. To create new tabs for your customized registers, use the `gui_tab` setting in the register definition (see “The GRD Register Definition Format” on page 610 for details).

The **Register View** window tab configuration files maintain settings across Debugger sessions. You will usually not need to modify these configuration files. If you are customizing the Debugger for internal distribution, however, you may want to add default tabs for use by your users in addition to the Processor, Board, and Ungrouped tabs. To add new default tabs, you would create a **Register View** window configuration file and install it in the configuration directories of your users. If you are not creating such customized in-house configurations, you do not need to be familiar with the contents of this section.

Default Loading of Configuration Files

Register View window configuration files are stored in a **regview** subdirectory inside your personal configuration directory (*user_dir*\Application Data\GHS\ on Windows and *user_dir*/.ghs/ on UNIX). Each time a root register description file is loaded by the Debugger, a corresponding configuration file is loaded from the **regview** directory. For a register description file named *foo.grd*, the corresponding window configuration file is named *foo.ini*. The default **.ini** configuration file describes the applicable default tabs. You can provide users with a new configuration by modifying or replacing the appropriate *foo.ini* configuration file.

File Syntax

The **Register View** window configuration file format is based on the generic utility options database.

When an entry named *grouppath* appears in keys in this section, it describes a full path to either a group or a register in dot-separated group path notation, which is the same as how groups are specified with the **regtab** command. For information about the **regtab** command, see “Register Commands” in Chapter 7, “Configuration Command Reference” in the *MULTI: Debugging Command Reference* book.

Expressions may not appear as values in pairs contained within a **Register View** window configuration file.

File Header

Each **Register View** window configuration file begins with a file header that identifies format of the file and gives settings that apply to the window as a whole instead of individual tabs. This header must be present for a file to be recognized as a valid **Register View** window configuration file. This header is contained in the top level package named *rwn*.

Version

A key-value pair must be present that gives the version of the window configuration file format. This version is given by a key-value pair of the form:

```
rwn.version=num
```

The version number should be 1.



Note This key-value pair is required to be present in every **Register View** window configuration file.

CPU Variant

When the default Processor and Board tabs are generated, an entry is added into the configuration file that contains a number identifying the exact type of target that the tabs are displaying. This is important because when you connect to a new type of target, the registers that are present on that target are usually different and the Processor and Board tabs need to be regenerated to reflect the new target. This identifier is stored in a key-value pair of the form:

```
rwn.cpuminor=num
```

If you are writing a set of default configuration files, you do not need to specify this key-value pair.

Describing Tabs

Each tab that appears in a **Register View** window is described in a corresponding top-level package *tab* within a **Register View** window configuration file. You can replace *tab* with any valid identifier except for the reserved top-level packages *rwn*, *~define*, *~sysinclude*, and *~include*.

Tab Name

Each tab is associated with a name that is used to describe it in the tab list at the top of a **Register View** window. A name is given to a tab by a key-value pair of the form:

```
tab.name=name
```

If this pair is omitted, *tab* is used as the tab's default name.

Root

A tab may be rooted at a node other than the default root of the register tree. A tab is given a different root by a key-value pair of the form:

```
tab.root=name
```

The value of this pair gives a fully qualified dot-separated group path to the group that should be used as the root of the tab.

If this key-value pair is omitted, the tab will display all of the groups of the register tree.

Index

The left to right ordering of tabs inside of the list at the top of a **Register View** window is saved in the configuration file through assigning an integer index to each tab. The tabs are then listed at the top of the window in order of increasing indices from left to right. An index is given to a tab by a key-value pair of the form:

```
tab.index=num
```

If this pair is omitted, the tab is given a default index of 0 and will appear on the left in the tab ordering.

Hidden Groups and Registers

The **Register View** window configuration file indicates which registers should be hidden in the tab display. Each hidden register or group has one corresponding entry in the configuration file. A hidden register or group that has a dot-separated group path *grouppath* is specified by a key-value pair of the form:

```
tab.hidden.grouppath=true
```

If the value of this pair is not `true`, then the corresponding register or group is displayed on the tab *tab*.

Collapsed Groups

The **Register View** window configuration file stores which groups were in the collapsed state the last time the window was closed. When the window is opened again, these collapsed groups will be displayed in the tree initially in the contracted state. A collapsed group that has a dot-separated group path *grouppath* is specified by a key-value pair of the form:

```
tab.collapsed.grouppath=true
```

If the value of this pair is not `true`, then the corresponding group is not initially collapsed on the tab.

This key-value pair has no effect for registers.

Index

(number sign) variable search
 designator, 273, 278
\$ (dollar sign) variable search
 designator, 277
* (asterisk) wildcard character, 284
. (period) variable search designator, 278
: (colon) variable search designator, 277
< (left angle bracket) command
 menu equivalent, 529
= (equal sign) operator, 274
> (right angle bracket) command
 menu equivalent, 528
? (question mark)
 wildcard character, 284
@ (at sign) wildcard character, 284
^ (caret) command, 280
> (greater than sign)
 in C++ classes, 174
-- (minus sign-double) command line
 option, 564
.* (period, asterisk) operator, 275
// (slash-double) comments, 275
:: (colon-double) variable search
 designator, 277
== (equal sign-double) operator, 274
>> (right angle bracket-double) command
 menu equivalent, 528 to 529
->* (minus sign, right angle bracket,
 asterisk) operator, 275
... Data Explorer message, 181
/* */ (slash, asterisk-double, slash)
 comments, 275

A

-a driver option, 343
About banner
 disabling with **-nosplash**, 566
About dialog box, 531
About MULTI menu item, 531
active group, 157, 159
address bus walking test, 404
addresses
 source, expressing in Debugger, 273
 viewing program at, 138, 516
AddressSpaces
 freeze-mode (OSA), 34
 profiling, 341, 345
 run-mode, 425
All Types menu item, 518
alternate directories
 ignoring with **-D**, 565
 specifying for search with **-I**, 24, 566
annotations, *see* Debugger Notes
ANSICMODE system variable, 291
any-task breakpoints
 freeze-mode, 462
 run-mode, setting, 430
arguments
 passing with -- (minus
 sign-double), 564
Arguments dialog box, 507
arrays in Data Explorer, 173
arrows in source pane, 40, 101
_ASMCACHE system variable, 293
assem command, 517
Assembly button, 534

Index

- assembly code
 - infinite scrolling, 130
 - interlaced with source code, 41
 - viewing in Debugger, 41
- Assembly Only** menu item, 517
- associating executable and connection
 - methods for, 87
 - on multi-core system, 465 to 466
- asterisk (*) wildcard character, 284
- at sign (@) wildcard character, 284
- Attach on Fork/Thread** menu item, 510
- Attach on Task Creation** menu item, 511
- Attach to Process** menu item, 500
- attaching to a process, 500
 - with **-P**, 24, 567
- attaching to a task, 425
- Auto Check Coherency** menu item, 510
- Auto Update Views** menu item, 515
- `_AUTO_CHECK_COHERENCY` system variable, 293
- `_AUTO_CHECK_NUM_ADDRS` system variable, 293
- B**
- beeping
 - in incremental Debugger search, 131
- bitfield definitions
 - changing, 256
- block detailed profiling report, 355
- board setup script, multi-core system, 464
- Bookmarks** menu item, 530
- `$bp_adr()` special operator, 298
- bpview** command, 105
- `_BREAK` system variable, 295
- breakdots, 40, 103
- breakpoints, 102
 - See also* **Breakpoints** window
 - any-task, freeze-mode, 462
 - any-task, run-mode, 430
 - deleting, 106
 - disabling, 104
 - enabling, 104
 - freeze-mode, 462
 - hardware, 110
 - See also* hardware breakpoints
 - information about, 105
 - jump, 548
 - listing, 518
 - markers, 40, 103
 - moving, *see* software breakpoints
 - overview, 102
 - run-mode, 430
 - run-mode, group, 431 to 432
 - setting basic, 40, 106
 - shared object, 113
 - software, 105
 - See also* software breakpoints
 - task-specific, 430, 462
 - toggling status of, 104
- Breakpoints** button, 105, 535
- Breakpoints** menu item
 - in **View** menu, 105, 512
 - in **View** → **List** submenu, 518
- Breakpoints** window, 105
 - See also* breakpoints
 - hardware breakpoints, 110, 112, 123, 125 to 127
 - opening, 105, 512
 - shared object breakpoints, 113, 123, 125 to 127
 - shortcut menus, 127
 - software breakpoints, 108, 123, 125 to 127
- browse** command, 362
- Browse** menu, 518
- Browse window
 - cross references, browsing, 216 to 219
 - data types, browsing, 214 to 216
 - filtering content in, 196 to 199
 - global variables, browsing, 209 to 211
 - headings, 200, 207, 213, 216
 - menus, 195

Index

- overview, 194
- procedure ambiguities, 207
- procedures, browsing, 142, 200, 202, 204 to 205, 207
- source files, browsing, 140, 211 to 213
- browsing
 - calls, classes, *see* Tree Browser
 - classes, includes, *see* Graph View
 - window
 - cross references, data types, global variables, procedures, source files, *see* Browse window
- build** command line option, 564
- Builder
 - debugging information, generating, 18
- Builder options
 - Profiling - Call Count**, 325, 343
 - Profiling - Coverage**, 343
 - Profiling - Target-Based Timing**, 342
 - Run-Time Memory Checks**, 321, 324 to 325
 - Shared Libraries**, 326
- buttons, 531
 - See also* toolbar
 - adding to toolbar, 538
 - commands for, 558
 - configuring, 538
 - in Debugger, 531
 - removing from toolbar, 538
 - reprogramming, 541
- C**
- c** command line option, 23, 564
- C++
 - classes in Data Explorer, 174 to 175
 - command line procedure calls, caveats for, 287
 - expressions and language dependencies, viewing, 284
 - expressions, caveats for, 275
 - unsupported operators, casts, destructors in expressions, 275, 287
- Cache Find** window, 383
- Cache Memory Reads** menu item, 510
- `_CACHE` system variable, 294
- Cache View** window, 379, 382
- caches
 - searching, 379, 383
 - viewing, 379, 382 to 383
- call count profiling data, 340 to 341, 343, 346
- call graph
 - profiling data, 340 to 341, 343
 - profiling report, 353
- Call Stack** button, 535
- Call Stack** menu item, 512
- Call Stack** window
 - buttons, 376
 - call stack pane, 377
 - call stack, viewing, 376
 - command line procedure calls, 377
 - functions, listing, 535
 - opening, 376, 512
 - procedure prologues and epilogues, 377
- calls
 - browsing, 193, 225 to 226
- caret (^) command, 280
- casts, in C++
 - not supported in expressions, 275
- caveats
 - for command line procedure calls, 286
 - for expressions, 274
 - for profiling, 346
- .cfg** files, ignoring with **-nocfg**, 566
- check=alloc** driver option, 321
- check=memory** driver option, 321
- classes
 - browsing, 193, 224
- Classes** menu item, 518
- Clear User Default Configuration** menu item, 526

Index

- clearing
 - profiling data, 370
- Close All Views** menu item, 515
- Close Debugger** button, 537
- Close Debugger Window** menu item, 501
- Close Entry** menu item, 500
- closing multiple windows, 515
- cmd** command line option, 564
- Cmd** pane, 45
- coherency checking, 417
- collecting
 - profiling data, 343 to 345
- colon (:) variable search designator, 277
- colon-double (::) variable search designator, 277
- command groups
 - register commands, 262
- command help, 8
- command line
 - starting MULTI from, 22, 564
- command line options
 - (minus sign-double), 564
 - build**, 564
 - c**, 23, 564
 - cmd**, 564
 - config**, 23, 564
 - configure**, 23, 564
 - connect**, 23, 564
 - connectfile**, 565
 - D**, 565
 - data**, 23, 565
 - display**, 23, 565
 - e**, 565
 - E**, 565
 - h**, 23, 565
 - H**, 23, 565
 - help**, 23, 565
 - I**, 24, 566
 - listing with **-h**, 23, 565
 - m**, 26, 566
 - minus sign-double (--), 564
 - nocfg**, 566
 - nodisplay** (UNIX), 24, 566
 - norc**, 24, 566
 - noshared**, 566
 - nosplash**, 566
 - notifylib**, 566
 - osa**, 451, 567
 - p**, 24, 567
 - P**, 24, 567
 - passing in a specification file, 25
 - See also* **-m** command line option
 - preload**, 567
 - r**, 24, 567
 - R**, 24, 567
 - rc**, 568
 - RO**, 25, 568
 - run**, 568
 - servertimeout**, 568
 - socket**, 568
 - text**, 25, 568
 - top**, 569
 - tv**, 569
 - usage**, 569
 - V**, 569
 - wmhints**, 569
- command line procedure calls
 - call stack for, 377
 - caveats for, 286
 - in Debugger, 285
 - unsupported class members, operators
 - in C++, 287
- command pane
 - Debugger, 45, 150
 - shortcut menus, 552
 - shortcuts, 47, 561
- command script, reading on startup, 568
- commands
 - < (left angle bracket)
 - menu equivalent, 529
 - > (right angle bracket)
 - menu equivalent, 528

Index

- ^ (caret), 280
 - >> (right angle bracket-double)
 - menu equivalent, 528 to 529
- assem**, 517
- b**, 107
- bpview**, 105
- browse**, 362
- caret (^), 280
- connect -restart_runmode**, 72
- dataview**, 577, 602
- Debugger button equivalents, 558
- dialogsearch**, 552
- dvclear**, 576
- dvload**, 576
- dvprofile**, 574, 600
- e**, 138
- edithwbp**, 111
- editswbp**, 107
- h**, 528
- help regarding, 8
- helpview**, 8
- hook, 464
- left angle bracket (<)
 - menu equivalent, 529
- memtest**, 414
- profdump**, 362, 367
- profile**, 344 to 345
- profilemode**, 358 to 363, 366, 368 to 370
- profilereport**, 350, 358
- recording to playback files, 24, 567
- restore**, 528
- right angle bracket (>)
 - menu equivalent, 528
- right angle bracket-double (>>)
 - menu equivalent, 528 to 529
- save**, 528
- sb**, 431, 462
- serialconnect**, 524
- serialdisconnect**, 525
- set_runmode_partner**, 72
- sethbp**, 111
- update**, 169
- view**, 164
- viewdel**, 164
- viewlist**, 172
- comments
 - in expressions, 275
- Compare** menu item, 522
- compare test, CRC, 412
- compiler driver options, *see* driver options
- compilers, Green Hills
 - generating debugging information, 18
- compressed target list display, 35
- compute test, CRC, 411
- concise display pane in **Register Information** window, 252
- conditionally not-executed
 - instructions, 101
- config** command line option, 23, 564
- Config** menu, 525
- configuration file, 23, 564
- configuration options
 - FirstPosition**, 180
 - help regarding, 8
 - MaxViewSize**, 180
 - MinViewSize**, 180
 - OSASwitchToUserTaskAutomatically**, 458
 - OSATaskAutoAttachLimit**, 457
 - ProcRelativeLines**, 42
- configure** command line option, 23, 564
- configuring
 - buttons, 538
 - line numbers in source pane, 42
 - MULTI with **-c**, 23, 564
 - toolbar, 538
- connect -restart_runmode** command, 72
- connect** command line option, 23, 564
- Connect** menu item, 519
- connected targets
 - managing from **Connection Organizer**, 80

Index

-connectfile command line option, 565

connecting

from **Connection Organizer**, 79

to debug server with **-connect**, 23, 564

to hardware (freeze mode) and OS (run mode), 70

troubleshooting, 74

to OS (run mode), 422

to process with **-P**, 24, 567

to serial port, *see* serial connections

to simulator, 70

to target

Custom Connection Method, 67

Standard Connection Method, 66

Temporary Connection Method, 69

Connection Chooser

creating Custom Connection

Method, 67

creating Standard Connection

Method, 58

opening, 58

Connection Editor

configuring Standard Connection

Method, 59

opening, 59

sections and fields of, 61

Connection Files, 78

Connection Methods

connecting with, 66, 79

creating, 77

Custom, 57, 67

editing, 59, 61, 77

overview, 56

Serial, 482, 484 to 485, 487

Standard, 57 to 59, 61

Temporary, 69

Connection Organizer

connected targets in, 80, 83

connecting to target from, 79

Connection Files in, 78, 80

Connection Methods in, 77, 81

opening, 76

context-sensitive help, 8

CONTINUECOUNT system variable, 291

CONTINUING status bar message, 54

conventions

typographical, xviii

Copy menu item, 521

copying text

in command pane (UNIX), 47, 561

in main Debugger window, 135

cores, multiple, *see* multi-core systems

coverage analysis profiling data, 340 to

341, 343, 346

coverage profiling report, 354

CPU % target list column, 38, 447

CRC compare test, 412

CRC compute test, 411

cross references

browsing, 193, 216 to 219

current line pointer, 40

Current PC menu item, 137, 516

`_CURRENT_TASKID` system

variable, 295

Custom Connection Methods, 57, 67

Customize Menus menu item, 525

Customize Toolbar menu item, 526

Customize Toolbar window, 538

cutting text in Debugger windows, 134

D

-D command line option, 565

data bus walking test, 406

-data command line option, 23, 565

data descriptions

MULTI data visualization (**.mdv**)

files, 574, 578

Data Explorer

activating variables, 166, 169

adding variables to, 167

arrays, viewing, 173

Index

- buttons, 166
- C++ classes, viewing, 174
- closing, 164
- custom, 572, 577
- dimensions, 178 to 179
- edit bar, 167
- freezing variables, 166, 169
- global options for, 178
- limiting data complexity, 178
- linked lists, viewing, 171
- messages, 181
- modifying variables in, 167, 177
- opening, 164, 168
- pointers, viewing, 173, 175
- structures, viewing, 170
- toolbar, 166
- types, changing, 175
- updating, 169
- viewing multiple items in, 168
- window, 165
- data offsets, position-independent data (PID)
 - specifying with **-data**, 23, 565
 - specifying with **_DATA**, 294
 - specifying with **Data Offset**, 93
- data pattern test, 408
- _DATA** system variable, 294
- data types
 - browsing, 193 to 194, 214 to 216
 - profiling, 340
- data visualization, *see* MULTI data visualization
- data visualization profile, 574, 600
- dataview** command, 577, 602
- .dbs** setup scripts, 63
- Dead Data Explorer message, 181
- Debug** menu, 503
- Debug Program as New Entry** menu item, 499
- Debug Program** menu item, 499
- debug servers
 - connecting to with **-connect**, 23, 564
 - logging connections, 62
 - setting default timeout with **-servertimeout**, 568
 - starting, 67, 69
 - use, 57
- Debug Settings** menu item, 506
- DebugBrk** target list status, 37
- Debugger, 6
 - See also* debugging
 - command line options, *see* command line options
 - connecting to your target, 56
 - graphical (GUI) mode, 20, 23, 565
 - menus, *see* Debugger menus
 - non-graphical (non-GUI) mode (UNIX), 21, 24, 566
 - notes in code, *see* Debugger Notes
 - overview, 6
 - searching source, 143, 552
 - special operators, 298
 - starting, 20
 - system variables, 291 to 292
 - version information, 569
 - window, *see* Debugger window
- Debugger Help** menu item, 530
- Debugger menus
 - Browse**, 518
 - Config**, 525
 - Debug**, 503
 - File**, 499
 - Help**, 530
 - Target**, 519
 - Tools**, 523
 - View**, 512
 - Windows**, 530
- Debugger Notes
 - browsing, 158
 - creating, 154
 - editing, 154, 156
 - grouping, 157 to 158

Index

- markers, 40
- overview, 154
- properties, 154
- Debugger Notes** menu item, 514
- Debugger window
 - assembly code, viewing, 41
 - breakdots, 40, 103
 - breakpoint markers, 103, 106
 - button-command equivalents, 558
 - buttons, 531
 - See also* buttons
 - navigation buttons, 142
 - closing, 501
 - command pane, 45, 150
 - controlling process from, 100
 - copying text, 135
 - current line pointer, 40
 - cutting text, 134
 - Debugger Note marker, 40
 - File Locator, 139
 - I/O** pane, 48
 - information pane, 31, 44
 - interlaced assembly code, viewing, 41
 - line numbers, 39, 42
 - menu bar, 31, 498
 - See also* Debugger menus
 - navigation bar, 31, 43
 - navigation buttons, 142
 - opening, 20
 - opening multiple, 32
 - output pane, 31, 44
 - overview, 30
 - pasting text, 134 to 135
 - PC pointer, 40, 101
 - Procedure Locator, 140
 - python pane, 50
 - reusing, 32
 - scroll bar with diamond (UNIX), 130
 - selecting text, 134 to 135
 - serial terminal pane, 49
 - shortcut menus, 543, 545, 548 to 552
 - shortcuts, 558
 - source code, viewing, 41
 - source pane, 31, 39, 136
 - See also* source pane
 - status bar, 32, 53
 - target list, 33
 - See also* target list
 - target pane, 48
 - toolbar, *see* toolbar
 - tracepoint markers, 103
 - traffic pane, 50
- debugging
 - disabling for shared libraries with
 - noshared**, 566
 - in freeze mode, 450
 - See also* freeze-mode debugging
 - memory allocation errors, 320
 - multi-core systems, 464
 - See also* multi-core systems
 - multiple files with **-E**, 565
 - multiple-language applications, 274
 - multitasking applications, 450
 - native processes, 378
 - operating systems, *see* freeze-mode debugging, *see* run-mode debugging
 - preliminary steps, 18
 - programs with specification files, 25
 - See also* **-m** command line option
 - in run mode, 422
 - See also* run-mode debugging
 - tasks, 426
- debugging information, 18
- Debugging Level
 - MULTI**, 285
- DEBUGSHARED system variable, 291
- default.con** file, 57
- Defines** menu item, 518
- Delete all view windows** button, 541
- DEREFPOINTER system variable, 291
- destructors in C++, 275
- Detach from Process** menu item, 500

Index

detailed display pane in **Register Information** window, 253
Dialog Boxes menu item, 518
dialog boxes, listing, 518
dialogsearch command, 552
diamond on Debugger scroll bar (UNIX), 130
Direct hardware access target list message, 36
directories, alternate
 ignoring with **-D**, 565
 specifying for search with **-I**, 24, 566
disabling
 profiling data collection, 347
Disassemble From Host menu item, 510
Disconnect from Serial menu item, 525
Disconnect from Target menu item, 84, 520
disconnecting
 from target, 84
DISNAMELEN system variable, 292
-display MULTI command line option, 23, 565
display problems, fixing, 569
DisplayMode menu item, 512
_DISPMODE system variable, 294
.dla symbol file, 18
.dnm symbol file, 18
document set, xvi to xvii
dollar sign (\$) variable search
 designator, 277
dots in source pane, 40, 103
Download Program button, 541
Download to RAM option, 95
downloading
 modules with **-preload**, 567
 separate executables on multi-core system, 466
 single executable on multi-core system, 465
 to RAM, 95

Downstack button, 535, 560
DownStack menu item, 136, 516
driver options
 -a, 343
 -check=alloc, 321
 -check=memory, 321
 -g, 42
 -G, 42, 285
 -lmulti, 285
 -p, 343
 -pg, 343
 -timer_profile, 342
Dump and Show Events button, 537
dumping profiling data, 367
dvclear command, 576
dvload command, 576
dvprofile command, 574, 600
DYING status bar message, 54
Dynamic Calls menu item, 519

E

e command, 138
-e command line option, 565
-E command line option, 565
edit bar
 in Data Explorer, 167
Edit button, 536
Editor menu item, 523
Editor, MULTI, 147
Empty Data Explorer message, 181
\$ent() special operator
 (deprecated), 298
\$entadr() special operator, 298
entry label, specifying with **-e**, 565
epilogue code
 debugging, 377
equal sign (=) operator, 274
equal sign-double (==) operator, 274
error checking, run-time, *see* run-time error checking

Index

evaluating

expressions, 272

examining data, *see* viewing

EXEC'ING status bar message, 54

_EXEC_NAME system variable, 295

Execing target list status, 37

\$exists() special operator, 299

Exit All menu item, 501

Exited target list status, 37

expressions

caveats for, 274

in Debugger commands, 272

evaluating, 272

formats for, 280

language-independent, 274

unsupported operators, casts,

destructors in C++, 275

viewing with wildcards, 284

F

F1 key, for help, 8, 530

Fast Source Step menu item, 511

File Calls menu item, 519

file chooser dialog box (UNIX), 554

File Locator on navigation bar, 139

File menu, 499

_FILE system variable, 295

file-relative line numbers

displaying in source pane, 39, 42

files

debugging multiple with -E, 565

listing, 517

recently opened from **File** menu, 500

searching, 133

Files menu item

in **Browse** menu, 518

in **View** → **List** submenu, 517

Fill menu item, 521

filtering content in

Browse window, 196 to 199

Find menu item, 521

find start/end ranges test, 412

FirstPosition configuration option, 180

formats for expressions, 280

free() standard library function

errors related to, 320

Freeze button

in Data Explorer, 166, 169

freeze-mode AddressSpaces, 34

freeze-mode breakpoints, 462

freeze-mode connections

concurrent with run-mode

connections, 70

troubleshooting, 74

freeze-mode debugging, 458

See also **OSA Explorer**

AddressSpaces, 34

breakpoints, 462

configuration file, 468

I/O, 463

in Debugger windows, 458 to 459

kernel, 457 to 458

Master Debugger mode, 458

multi-core systems, 464

See also multi-core systems

overview, 450

starting, 451

target environments supported, 450

Task Debugger mode, 459

task-specific single-stepping, 460

tasks, 457 to 459

freeze-mode tasks, 459

functions, *see* procedures

G

-g driver option, 18, 42

-G driver option, 18, 42, 285

generating profiling data, 342

global variables

browsing, 194, 209 to 211

Index

- listing, 517
- Globals** menu item
 - in **Browse** menu, 518
 - in **View** → **List** submenu, 517
- Go Back** button, 531
- Go on Selected Items** button, 533, 558
- Go on Selected Items** menu item, 503
- Go to next selected location in trace data** button, 541
- Go to previous selected location in trace data** button, 542
- Goto Location** menu item, 138, 516
- Graph View window
 - controlling layout in, 228
 - custom, 572, 577
 - history buttons, 231
 - includes, browsing, 228
 - Layout** menu, 229
 - navigating, 228
 - Search for Objects** window, 230
 - shortcut menus, 229
 - window, 227
- graphical (GUI) mode
 - starting Debugger in, 20, 23, 565
- GRD register definition files
 - format, 610
 - location of, 266
 - saving, 256
- greater than sign (>)
 - in C++ classes, 174
- Green Hills compilers
 - generating debugging information, 18
- grep** utility, 134
 - See also* searching
 - licensing, 134
- group breakpoints, run-mode, 431 to 432
- GrpHalt** target list status, 37
- H**
- h** command, 528
- h** command line option, 23, 565
- H** command line option, 23, 565
- Halt on Selected Items** button, 533
- Halt on Selected Items** menu item, 504
- Halt System** menu item, 504
- Halted** target list status, 37
- halting tasks on attach, 426
- hardware breakpoints
 - editing, 110
 - managing, 112, 123, 125 to 127
 - overview, 110
 - properties of, 112, 115
 - setting, 110
 - viewing, 112, 123, 125 to 127
- headings
 - in Browse window, 200, 207, 213, 216
- help, *see* online help
- help** command line option, 23, 565
- Help** menu, 530
 - online help, obtaining, 8
- Help Viewer
 - navigating, 10 to 12
 - opening additional manuals in, 13
 - window, 9
- helpview** command, 8
- hierarchical target list display, 35
- hook commands in multi-core setup
 - scripts, 464
- HostIO** target list status, 37
- I**
- I** command line option, 24, 566
- I/O
 - program, with INTEGRITY, 463
 - redirection, 503
- I/O pane, 48
- ignoring
 - alternate directories with **-D**, 565
 - .cfg** files with **-nocfg**, 566
 - .rc** files with **-norc**, 24, 566

Index

`$in()` special operator, 299
Included Files menu item, 518
includes
 browsing, 228
Includes menu item, 519
incremental searching
 beeping in, 131
 for strings, 130
infinite scrolling, 130
information pane, 31, 44, 441
_INIT_SP system variable, 294
input/output pane, 48
input/output redirection, 503
instructions
 conditionally not executed, 101
 dimmed, 101
 stepping through, 504
instrumented profiling, 342
Integrated Development Environment (IDE), *see* MULTI Integrated Development Environment (IDE)
INTEGRITY
 freeze-mode debugging, 450
 program I/O in freeze mode, 463
INTEGRITY Object Viewer button, 537
_INTERLACE system variable, 295
interlaced assembly code
 viewing in Debugger, 41
Interlaced Assembly menu item, 517

J
jump breakpoints, setting, 548

K
kernel
 debugging in freeze mode, 457 to 458
keyboard shortcuts, *see* shortcuts
keywords, language, 274
Kill Selected Items menu item, 504

L
language dependencies, 284
language keywords, 274
_LANGUAGE system variable, 294
language-independent expressions, 274
_LAST_COMMAND_STATUS system variable, 295
_LAST_CONNECT_CMD_LINE system variable, 295
Launch Utility Programs menu item, 524
Launcher, *see* MULTI Launcher
Launcher menu item, 523
left angle bracket (<) command
 menu equivalent, 529
libmulti.a file, 285
lifetime information for variables, 278
line numbers
 displaying in source pane, 39, 42
 jumping to with line pointer, 40
 memory address of, 273
 program counter, 40, 101
line pointer, 40
_LINE system variable, 295
_LINES system variable, 294
linked lists in Data Explorer, 171
List menu item, 515
listing
 breakpoints, 518
 command line options with **-h**, 23, 565
 dialog boxes, 518
 files, 517
 global variables, 517
 included source files, 518
 local variable addresses, 517
 local variables, 517
 macros, 518
 mangled procedures, 517
 procedure variables, 518
 procedures, 517
 processes, 518
 register synonyms, 518

Index

- registers, 517
- signals, 518
- source paths, 518
- special variables, 518
- static variables, 517
- lists
 - selecting items from, 135
- Imulti** driver option, 285
- Load Configuration** menu item, 526
- Load Module** menu item, 520
- Load Symbols** menu item, 506
- loading, *see* downloading
- Local Addresses** menu item, 517
- local variables, 513, 517
- Local Variables** menu item, 513
- Locals (\$locals\$) window, 145
- Locals** button, 536
- Locals** menu item, 517
- Lock Other Threads on Single Step** menu item, 512
- logging
 - debug server connections, 62
- M**
- m** command line option, 26, 566
- `$M_called_from()` special operator, 299
- `$M_can_read_address()` special operator, 299
- `$M_file_exists()` special operator, 299
- `$M_get_temp_memory()` special operator, 299
- `$M_offsetof()` special operator, 299
- `$M_sec_begin()` special operator, 300
- `$M_sec_end()` special operator, 300
- `$M_sec_exists()` special operator, 300
- `$M_sec_size()` special operator, 300
- `$M_strcaseprefix()` special operator, 300
- `$M_strcmp()` special operator, 300
- `$M_strcpy()` special operator, 300
- `$M_strprefix()` special operator, 300
- `$M_strstr()` special operator, 300
- `$M_sym_exists()` special operator, 299
- `$M_typeof()` special operator, 300
- `$M_var_is_valid()` special operator, 301
- macros
 - calling, 288
 - evaluating, 288
 - listing, 518
 - shortcut menu, 551
- Make Serial Connection** menu item, 524
- `malloc()`
 - errors related to, 320
- Manage Bookmarks** menu item, 530
- Mangled Procedures** menu item, 517
- mangled procedures, listing, 517
- Manuals** menu item, 530
- Master Debugger mode, 458
- master process, OSA, 458
- MaxViewSize** configuration option, 180
- .mbs** setup scripts, 63
- .mdv** files, *see* MULTI data visualization (**.mdv**) files
- memory
 - accessing from **Memory View** window, 308
 - allocation errors, *see* memory allocation
 - coherency, 417
 - editing from **Memory View** window, 309
 - leaks, finding, 320, 332
 - manipulating in **Memory Allocations** window, 334
 - preceding memory location, examining, 280
 - testing, *see* memory testing
 - viewing, 304

Index

- See also* **Memory View** window
 - from Data Explorer, 164
- memory allocation
 - debugging, 320
 - disabling, 322 to 323
 - enabling, 322 to 323
 - error checking, 320 to 323
 - tracking, 336
 - viewing, 328, 332
 - window, *see* **Memory Allocations** window
- Memory Allocations** menu item, 513
- Memory Allocations** window
 - enabling error-checking from, 323
 - leaks, viewing, 332
 - menus, 327
 - opening, 326, 513
 - overview, 325
 - procedures, manipulating, 334
 - refreshing, 335
 - tabs, 327
 - tracking leaks and allocations, 336
 - updating, 332
 - visualization, 328
- Memory** button, 535
- Memory Dump** menu item, 522
- Memory Load** menu item, 522
- Memory Manipulation** menu item, 521
- Memory** menu item, 513
- Memory Sensitive** menu item, 510
- memory sensitive mode, 97
- Memory Test** menu item, 521
- Memory Test Results** window, 402
- Memory Test Wizard**, 390
- memory testing
 - address bus walking test, 404
 - advanced, 394
 - coherency errors, 417
 - command line, 414
 - continuous, 403
 - CRC compare test, 412
 - CRC compute test, 411
 - data bus walking test, 406
 - data pattern test, 408
 - find start/end ranges test, 412
 - hints for efficient testing, 414
 - memory read test, 411
- Memory Test Results** window, 402
- Memory Test Wizard**, 390
- overview, 390
- Perform Memory Test** window, 394 to 395
 - See also* **Perform Memory Test** window
 - quick, 390
 - results, viewing, 402
 - running, 401
 - selecting, 397
 - with target agent, 403
 - test area, specifying, 395
 - test method, specifying, 400
 - test options, setting, 399
 - types, 404
- Memory View** window
 - active location, setting, 305
 - columns, 314
 - editing memory from, 309
 - formatting columns, 306
 - freezing, 308
 - infinite scrolling, 130
 - memory pane, 306
 - opening, 304, 513
 - overview, 304
 - updating, 308
- memtest** command, 414
- menu bar, 31, 498
 - See also* Debugger menus
- menus
 - Debugger, *see* Debugger menus
- messages
 - Data Explorer, 181
 - status bar, 53

Index

- minus sign, right angle bracket, asterisk
 (->*) operator, 275
- minus sign-double (--) command line
 option, 564
- MinViewSize** configuration option, 180
- Modify Register Definition** dialog, 256
 to 258
- modules, downloading with **-preload**, 567
- Move splitter down** button, 33
- Move splitter left** button, 33
- Move splitter right** button, 33
- Move splitter up** button, 33
- Move target list to left** button, 33
- moving software breakpoints, 109
- mterminal** command, 492
- MTerminal** serial terminal emulator, 482
 - See also* serial connections
 - configuring, 484 to 485, 487
 - console mode (UNIX only), 492
 - creating Connection Methods, 484 to 485, 487
 - overview, 482
 - as stand-alone application, 492
 - starting, 482
 - window features, 488
- MULTI, *see* MULTI Integrated Development Environment (IDE)
- MULTI Builder, *see* Builder
- MULTI data visualization
 - with Data Explorer, 171
 - invoking, 577
 - MULTI data visualization (**.mdv**)
 - files, *see* MULTI data visualization (**.mdv**) files
 - overview, 572
- MULTI data visualization (**.mdv**) files
 - clearing, 576
 - data descriptions, 574, 578
 - expressions in, 604
 - file format, 578
 - loading, 576
 - overview, 572, 574
 - profile descriptions, 574, 600
 - type-specific fields, 581 to 596
 - view descriptions, 574, 602
- MULTI Debugger, *see* Debugger
- MULTI Debugging Level**, 285
- MULTI Editor, 147
- MULTI EventAnalyzer** button, 537
- MULTI EventAnalyzer** menu item, 523
- MULTI Integrated Development Environment (IDE)
 - document set, xvii
 - exiting, 501
 - online help, 8
 - overview, 2
 - starting for Object Structure Awareness (OSA), 567
 - starting from command line, 22, 564
 - viewing information about, 531
- MULTI Launcher
 - overview, 4
- MULTI ResourceAnalyzer** button, 537
- MULTI ResourceAnalyzer** menu item, 523
- MULTI Variables** menu item, 518
- multi-core systems
 - display in target list, 464
 - hook commands in setup script, 464
 - limitations, 468
 - running separate executables on, 466
 - running single executable on, 465
 - synchronous run control, 467
- _MULTI_DIR system variable, 296
- _MULTI_MAJOR_VERSION system variable, 296
- _MULTI_MICRO_VERSION system variable, 296
- _MULTI_MINOR_VERSION system variable, 296
- multiple files, debugging with **-E**, 565
- multiple run-mode connections, 422

Index

multiple-language applications,
 debugging, 274
multitasking applications, debugging, 450

N

NaN Data Explorer message, 181
native processes
 viewing, 378
navigating
 buttons for, 142
 MULTI windows, 130
navigation bar, 31, 43
Navigation menu item, 512
Next (over Functions) on Selected Items
 button, 532, 559
Next (over Functions) on Selected Items
 menu item, 504
No process Data Explorer
 message, 181
NO PROCESS status bar message, 54
No Stack Trace menu item, 510
no stack trace mode, 98
No symbols for this
 procedure Data Explorer
 message, 181
No TimeMachine Data target list
 status, 37
-nocfg command line option, 566
node operations in Tree Browser, 222
-nodisplay MULTI command line option
 (UNIX), 24, 566
non-graphical (non-GUI) mode (UNIX)
 starting Debugger in, 21, 24, 566
_NONGUIMODE system variable, 296
-norc command line option, 24, 566
-noshared command line option, 566
-nosplash command line option, 566
Not Initialized Data Explorer
 message, 181
Not loaded target list status, 37

Note Browser, 158
Notepad menu item, 523
-notifylib command line option, 566
Number of Addresses to Check menu
 item, 511
number sign (#) variable search
 designator, 273, 278

O

Object Structure Awareness (OSA), 451
 See also **OSA Explorer**
 starting MULTI for, 451, 567
offsetof() special operator, 299
offsets
 position-independent code (PIC), 25,
 93, 295, 568
 position-independent data (PID), 23,
 93, 294, 565
online help, 8
 See also Help Viewer
 for commands, 8
 for configuration options, 8
 context-sensitive, 8
 limitations, 12
 online manuals, 8
 opening with **-help**, 23, 565
 opening with F1, 530
 overview, 8
_OPCODE system variable, 295
Open project manager for current
 project button, 542
operating systems
 debugging, *see* freeze-mode debugging,
 see run-mode debugging
operators
 language-independent, 274
 not supported in expressions, 275, 287
 overloaded in C++, 285
 special, *see* special operators

Index

Optimized away Data Explorer
 message, 181
options, *see* command line options, *see*
 driver options
Options menu item, 525
Original procedure not on
 stack Data Explorer message, 181
OS-awareness, 446, 450 to 451, 468
 See also freeze-mode debugging; **OSA**
 Explorer; OSA Object Viewer
OSA AddressSpaces in target list, 34
-osa command line option, 451, 567
OSA Explorer, 452
 See also Object Structure Awareness
 (OSA)
 displaying, 452
 menus, 455
 object list, 456
 opening, 514
 overview, 452
 shortcut menus, 456, 475
 toolbar, 455
OSA Explorer button, 536
OSA Explorer menu item, 514
OSA master process, 458
OSA Object Viewer, 446
OSA tasks, 459
OSASwitchToUserTaskAutomatically
 configuration option, 458
OSATaskAutoAttachLimit configuration
 option, 457
output
 recording to playback files, 24 to 25,
 567 to 568
output pane, 31, 44
output/input redirection, 503

P

-p command line option, 24, 567
-P command line option, 24, 567
-p driver option, 343

pane-switching tabs, 31, 44
partner, run-mode, *see* run-mode partner
passing arguments with **--** (minus
 sign-double), 564
pasting text
 in Debugger windows, 134
 in main Debugger window, 135
PC (program counter), 40
 dimmed, 101
 samples, 340 to 342, 346
Pended target list status, 37
Perform Memory Test window, 394
 running tests from, 401
 test area, setting, 395
 test method, specifying, 400
 test options, setting, 399
 test type, selecting, 397
period (.) variable search designator, 278
period, asterisk (.*) operator, 275
-pg driver option, 343
 _PID system variable, 296
Playback Commands menu item, 529
playback files, 24 to 25, 567 to 568
pointers in Data Explorer, 173, 175
position-independent code (PIC)
 offset, specifying with **-text**, 25, 568
 offset, specifying with **_TEXT**, 295
 offset, specifying with **Text Offset**, 93
position-independent data (PID)
 offset, specifying with **-data**, 23, 565
 offset, specifying with **_DATA**, 294
 offset, specifying with **Data Offset**, 93
-preload command line option, 567
Prepare Target button, 534
Prepare Target dialog box, 91
Prepare Target menu item, 506
preparing a target
 basics, 90
 by downloading, 95
 by verifying, 95
 multi-core, 465 to 466

Index

- settings related to, 97 to 98
- Previous** button, 532
- Print Expression** menu item, 514
- Print** menu item, 499
- Print Setup** dialog box (UNIX), 501
- Print Window** menu item, 499
- printing
 - in UNIX, 501
- Procedure Locator on navigation bar, 140
- `_PROCEDURE` system variable, 296
- procedure-relative line numbers
 - displaying in source pane, 39, 42
- procedures
 - ambiguities, Browse dialog box for, 207
 - browsing, *see* Browse window
 - calling from command line, 286
 - calling from Debugger, 285
 - epilogues, 377
 - listing, 517
 - mangled, listing, 517
 - manipulating in **Memory Allocations** window, 334
 - in Procedure Locator, 140
 - prologues, 377
 - shortcut menu for, 548
 - stepping into, out of, over, 504
- Procedures in Files** menu item, 519
- Procedures** menu item
 - in **Browse** menu, 518
 - in **View** → **List** submenu, 517
- Process running Data Explorer message, 181
- `_PROCESS` system variable, 296
- Process Viewer**
 - opening, 514, 569
 - window, 378
- process-specific breakpoints, *see* task-specific breakpoints
- processes
 - controlling from Debugger, 100
 - halting, 504
 - killing current, 504
 - listing, 518
 - native, viewing, 378
 - sending signal to, 505
 - state of, 53
 - stopping, 100
- Processes** menu item, 518
- processing profiling data, 368
- ProcRelativeLines** configuration
 - option, 42
- profdump** command, 362, 367
- profile** command, 344 to 345
- profile descriptions
 - MULTI data visualization (**.mdv**) files, 574, 600
- Profile** menu item, 513
- Profile** window, 348
 - See also* profiling reports
 - menus, 357 to 358
 - overview, 348
 - toolbar, 360
 - updates in, 349
- profilemode** command, 358 to 363, 366, 368 to 370
- profilereport** command, 350, 358
- profiling, 341
 - See also* profiling data
 - AddressSpaces, 341, 345
 - caveats, 346
 - instrumented, 342
 - methods overview, 341
 - PC sampling, 341 to 342, 447
 - with **protrans**, 371
 - range, 364
 - stand-alone program, 341, 345
 - target-based timing, 342
 - tasks
 - all in system, 341, 344, 447
 - single, 341 to 342, 345 to 346

Index

- trace-enabled target, 341
 - Profiling - Call Count Builder**
 - option, 325, 343
 - Profiling - Coverage Builder** option, 343
 - Profiling - Target-Based Timing Builder**
 - option, 342
 - profiling data, 340
 - See also* profiling; profiling reports
 - adding to, 366
 - call count, 340 to 341, 343, 346
 - call graph, 340 to 341, 343
 - clearing, 370
 - collecting, 343 to 345
 - coverage analysis, 340 to 341, 343, 346
 - disabling collection of, 347
 - dumping, 367
 - instrumenting code to generate, 342
 - overwriting, 366
 - PC (program counter) samples, 340 to 342, 346
 - processing manually, 368
 - types of, 340
 - viewing, 347, 350, 363
 - profiling reports
 - block detailed, 355
 - call graph, 353
 - coverage, 354
 - overview, 350
 - source, 356
 - standard calls, 352
 - status, 351
 - Program already present on target.**
 - Verify** option, 95
 - program counter (PC), 40
 - dimmed, 101
 - samples, 340 to 342, 346
 - Program Counter** button, 534
 - programs
 - execution of, 100
 - profiling stand-alone, 341, 345
 - RAM download, 93
 - ROM copy, 94
 - ROM run, 93
 - starting, 100
 - stepping through, 101
 - unknown, 94
 - Project Manager
 - opening, 523, 542
 - Project Manager** menu item, 523
 - prologue code
 - debugging, 377
 - protrans** utility, 371
 - Py** pane, 50
 - python pane, 50
- ## Q
- question mark (?) wildcard character, 284
- ## R
- r** command line option, 24, 567
 - R** command line option, 24, 567
 - RAM download programs, 93
 - range analysis, 364
 - rc** command line option, 568
 - .rc** files
 - customizing registers with, 265
 - ignoring with **-norc**, 24, 566
 - read-only system variables, 295
 - reading command scripts on startup, 568
 - Rebuild** menu item, 523
 - Record Command+Output** menu item, 528
 - Record Commands** menu item, 528
 - recording commands
 - menu items for, 528 to 529
 - to playback files, 24, 567
 - recording output
 - to playback files, 24 to 25, 567 to 568
 - redirecting input/output, 503
 - Refresh Views** menu item, 515
 - register definition files

Index

- customizing, 265, 268 to 269
- default, 265, 268 to 269
- GRD format, 256, 610
- location of, 266
- overloading, 268
- overview, 610
- searching for, 266
- Register Explorer, 234
 - See also* register definition files;
 - Register Information** window;
 - Register View** window
- overview, 234
- startup, 266
- Register Information** window
 - buttons, 255
 - opening, 251
 - overview, 250 to 251
 - panes, 252 to 254
 - register values, changing, 255
 - shortcut menus, 252, 254
- Register Search** window, 260
- Register Setup** dialog, 259 to 260
- Register Synonyms** menu item, 518
- Register View** window
 - configuration files, 249, 634 to 638
 - copying register values, 247
 - customizing, 243 to 246, 634
 - editing register contents, 247
 - File** menu, 236
 - Format** menu, 238
 - opening, 234, 513
 - overview, 234 to 235
 - printing window contents, 249
 - refreshing, 248
 - register tree, 241, 246 to 247
 - shortcut menus, 242
 - tabs, 240, 244 to 246
 - toolbar, 239
 - View** menu, 237
- registers
 - bitfield definitions, 256
 - changing values of, 255
 - commands, 262
 - copying values, 247
 - creating, 259 to 260
 - customizing, 265
 - defining, *see* register definition files
 - editing contents, 247
 - listing, 517
 - printing list of, 249
 - searching for, 260
 - viewing, *see* **Register Information** window, *see* **Register View** window
- Registers** button, 536
- Registers** menu item
 - in **View** menu, 513
 - in **View** → **List** submenu, 517
- Reload** button, 534
- `_REMOTE_CONNECTED` system variable, 296
- Remove All Breakpoints** menu item, 505
- reports, profiling, *see* profiling reports
- reprogramming
 - buttons, 541
- rerooting Tree Browser, 223
- Reset** button, 534
- Restart** button, 533, 559
- Restart** menu item, 503
- restore** command, 528
- Restore State** menu item, 528
- `$result` special predefined variable, 298
- `$ret()` special operator
 - (deprecated), 301
- `$retadr()` special operator, 301
- Return on Selected Items** button, 533, 559
- Return on Selected Items** menu item, 504
- reusing windows, 32
- right angle bracket (`>`) command
 - menu equivalent, 528
- right angle bracket-double (`>>`) command
 - menu equivalent, 528 to 529

Index

- right-click
 - in command pane, 561
- RO** command line option, 25, 568
- ROM copy programs, 94
- ROM run programs, 93
- routines, *see* procedures
- `_RTSERV_VER` system variable, 296
- run** command line option, 568
- run control, synchronous
 - on multi-core systems, 467
- Run System** menu item, 505
- Run to Cursor** menu item, 504
- run-mode AddressSpaces, 425
- run-mode breakpoints
 - any-task, setting, 430
 - group, 431 to 432
 - task-specific, 430
- run-mode connections
 - automatically established, *see* run-mode partner
 - concurrent with freeze-mode
 - connections, 70
 - troubleshooting, 74
 - multiple, 422
 - overview, 422
- run-mode debugging, 424
 - See also* Task Manager
 - AddressSpaces, 425
 - attaching to tasks, 425
 - breakpoints, 430
 - overview, 422
- run-mode partner
 - disabling, 74
 - overview, 71
 - setting, 72
 - troubleshooting, 74
- run-mode tasks
 - attaching to, 425
- run-time error checking
 - enabling and disabling, 322 to 323
 - generating information for, 19

- memory allocation, 320
- Run-Time Memory Checks** Builder
 - option, 321, 324 to 325
- RUNNING status bar message, 54
- Running** target list status, 37

S

- save** command, 528
- Save Configuration As** menu item, 526
- Save Configuration as User Default** menu item, 526
- Save State** menu item, 528
- sb** command, 431, 462
- scripts
 - command, reading on startup, 568
 - .dbs**, 63
 - .mbs**, 63
 - syntax checking, 301
- scroll bars
 - diamond in (UNIX), 130
- scrolling
 - infinite, 130
- Search for Objects** window, 230
- Search in Files** menu item, 524
- Search** menu item, 524
- searching
 - beeping in incremental, 131
 - caches, 383
 - files, 133
 - incrementally, 130
 - MULTI windows, 130
 - shortcuts for, 131, 553
 - source pane, 143, 552
 - with variable search designators, 277
 - with wildcards, 284
- selecting
 - items from lists, 135
 - text in Debugger windows, 134
 - text in main Debugger window, 135
- `_SELECTION` system variable, 296

Index

- Send Signal** menu item, 505
- Serial Connection Chooser**, 484
- Serial Connection Settings** dialog, 485, 487
- serial connections, 482
 - See also MTerminal* serial terminal emulator
 - configuring, 484 to 485, 487
 - creating, 484 to 485, 487
 - overview, 482
 - starting, 482
- Serial Terminal** menu item, 524
- serial terminal pane, 49
- serialconnect** command, 524
- serialdisconnect** command, 525
- ServerPollinterval** option, 448
- servertimeout** command line option, 568
- SERVERTIMEOUT system variable, 292
- Set And Edit** menu item, 548
- Set Any Task Breakpoint** menu item, 548
- Set Breakpoint** menu item, 548
- Set INTEGRITY Distribution** menu item, 527
- Set Jump Breakpoint** menu item, 548
- Set Program Arguments** menu item, 503
- Set Run-Mode Partner** dialog box, 72
- Set Run-Mode Partner** menu item, 72, 74, 520
- Set u-velOSity Distribution** menu item, 527
- Set Watchpoint** menu item, 505
- set_runmode_partner** command, 72
- setup script, multi-core system, 464
- shared libraries
 - disable debugging for with **-noshared**, 566
- Shared Libraries** Builder option, 326
- shared object breakpoints, 113
 - managing, 113, 123, 125 to 127
 - viewing, 113, 123, 125 to 127
- shortcuts
 - command pane, 47, 561
 - Debugger window, main, 558
 - search, 131, 553
 - source pane, 560
- Show Command History** menu item, 528
- signals
 - listing, 518
 - sending to current process, 505
- Signals** menu item, 518
- simulators
 - connecting to, 70
- single-stepping
 - process, 101
 - task-specific, in freeze mode, 460
- sizeof()** special operator, 301
- slash, asterisk-double, slash (**/ * */**)
 - comments, 275
- slash-double (**//**) comments, 275
- socket** command line option, 568
- software breakpoints
 - editing, 106
 - managing, 108, 123, 125 to 127
 - moving, 109
 - overview, 105
 - properties of, 115
 - setting, 106
 - viewing, 108, 123, 125 to 127
- source addresses
 - expressing in Debugger, 273
- source code
 - interlaced with assembly code, 41
 - stepping through, 504
 - viewing in Debugger, 41
- source files
 - browsing, 140, 194, 211 to 213
 - listing, 518
- source lines
 - shortcut menu for, 545
- Source Only** menu item, 517
- source pane
 - browsing, 136

Index

- content displayed in, 34
- display modes, 41
- navigating, 136
- overview, 31, 39
- searching, 136, 143, 552
- shortcut menus, 545
- shortcuts, 545, 560
- Source Path** menu item, 515
- source paths
 - listing, 518
- Source Paths** menu item, 518
- source profiling report, 356
- special operators
 - `$bp_adr()`, 298
 - `$ent()` (deprecated), 298
 - `$entadr()`, 298
 - `$exists()`, 299
 - `$in()`, 299
 - `$M_called_from()`, 299
 - `$M_can_read_address()`, 299
 - `$M_file_exists()`, 299
 - `$M_get_temp_memory()`, 299
 - `$M_offsetof()`, 299
 - `$M_sec_begin()`, 300
 - `$M_sec_end()`, 300
 - `$M_sec_exists()`, 300
 - `$M_sec_size()`, 300
 - `$M_strcaseprefix()`, 300
 - `$M_strcmp()`, 300
 - `$M_strcpy()`, 300
 - `$M_strprefix()`, 300
 - `$M_strstr()`, 300
 - `$M_sym_exists()`, 299
 - `$M_typeof()`, 300
 - `$M_var_is_valid()`, 301
 - `offsetof()`, 299
 - `$ret()` (deprecated), 301
 - `$retadr()`, 301
 - `sizeof()`, 301
- special variables
 - listing, 518
 - `$result`, 298
- specification files
 - specifying with **-m**, 566
 - using, 25
- splash screen
 - disabling with **-nosplash**, 566
- `Srch` status bar message, 53, 130
- Srl** pane, 49
- stand-alone program
 - profiling, 341, 345
- stand-alone windows, 144
- standard calls profiling report, 352
- Standard Connection Methods, 57 to 59, 61
- starting
 - Debugger, 20
 - in graphical (GUI) mode, 20, 23, 565
 - in non-graphical (non-GUI) mode (UNIX), 21, 24, 566
 - MULTI for Object Structure Awareness (OSA), 567
 - MULTI from command line, 22, 564
- State** menu item, 527
- state of process, 53
- `__STATE` system variable, 297
- Static Calls** menu item, 519
- static variables
 - listing, 517
- Statics** menu item, 517
- status bar
 - in Debugger window, 32
- status bar messages, 53
- status profiling report, 351
- Status** target list column, 37
- Step (into Functions) on Selected Items**
 - button, 532, 559
- Step (into Functions) on Selected Items**
 - menu item, 504
- Step Back** button, 532
- Step Up** button, 531

Index

Stop on Task Creation menu item, 511
Stop Record Commands+Output menu item, 529
Stop Recording Commands menu item, 528
stop sign in source pane, 106
STOPPED arrow (red) in Debugger, 40
 dimmed, 101
STOPPED INSIDE status bar message, 54
STOPPED status bar message, 54
Stopped target list status, 37
strings
 searching incrementally for, 130
structures in Data Explorer, 170
subroutines
 stepping, 504
symbol files for debugging, 18
synchronous run control, multi-core systems, 467
syntax checking
 commands, 301
 scripts, 301
SysHalt target list status, 37
system variables
 in Debugger, 291 to 292
 listing, 518
 read-only, 295

T

target list
 auto-sizing of, 33
 compressed display, 35
 CPU % column, 38, 447
 grouping of items in, 35
 hiding, 33
 hierarchical display, 35
 identifying items in, 34
 multiple cores in, 464
 opening multiple Debugger windows from, 32

 reusing Debugger window with, 32
 shortcut menu, 542
 showing, 33
 Status column, 37
 terminology, 36
Target menu, 519
target pane, 48
Target Settings menu item, 506
 _TARGET system variable, 297
target-based timing profiling, 342
 _TARGET_COPROCESSOR system variable, 297
 _TARGET_FAMILY system variable, 297
 _TARGET_IS_BIGENDIAN system variable, 297
 _TARGET_MINOR_OS system variable, 297
 _TARGET_OS system variable, 297
 _TARGET_SERIES system variable, 297
targets
 concurrent freeze-/run-mode
 connections to, 70
 troubleshooting, 74
 connecting to, 56
 disconnecting from, 84
 multi-core, *see* multi-core systems
 preparing, *see* preparing a target
 simulated, 70
 trace-enabled, profiling, 341
Task Debugger mode, 459
task groups, 428
 See also Task Manager; tasks
 breakpoints for, 431
 configuration file for, 444
 creating, 428
 default, 428
 synchronous operations, relating to, 432
Task Manager, 424
 See also task groups; tasks
 attaching to tasks, 425

Index

- debugging tasks, 426
- freezing task list, 427
- halting tasks, 425 to 426
- information pane, 441
- killing tasks, 425
- menus, 435 to 438
- mouse operations, 442
- opening, 424, 514
- overview, 424, 435
- running tasks, 425
- shortcut menus, 442
- task groups in, 428
- task list pane, 440
- toolbar, 439
- `_TASK_EXIT_CODE` system variable, 297
- task-specific breakpoints, 430
 - freeze-mode, 462
- task-specific single-stepping, freeze-mode, 460
- tasks, 422
 - See also* task groups; Task Manager
 - breakpoints for, 430, 462
 - displaying in the Debugger source pane, 426
 - freeze mode, debugging, 457 to 459
 - halting, 425
 - halting on attach, 426
 - killing, 425
 - profiling
 - all, 341, 344, 447
 - single, 341 to 342, 345 to 346
 - run-mode, attaching to, 425
 - running, 425
 - in target list, 457
 - terminology conventions, 422, 451
- Tasks** menu item, 514
- `TASKWIND` system variable, 292
- Temporary Connection Methods, 69
- tests, memory, *see* memory testing
- text** command line option, 25, 568
- text offsets, position-independent code (PIC)
 - specifying with **-text**, 25, 568
 - specifying with `_TEXT`, 295
 - specifying with **Text Offset**, 93
- `_TEXT` system variable, 295
- text, selecting, cutting, and pasting, 134
- Tfc** pane, 50
- threads, 451
- ThreadX
 - freeze-mode debugging, 450
 - `_tx_thread_created_ptr` OS-specific symbol, 451
- timeout
 - setting for debug servers, 568
- timer_profile** driver option, 342
- Toggle All Breakpoints** menu item, 505
- Toggle IO Buffering** menu item, 520
- toggling
 - breakpoint status, 104
 - tracepoint status, 104
- toolbar
 - adding buttons to, 538
 - button descriptions, 531
 - removing buttons from, 538
- Tools** menu, 523
- top** command line option, 569
- trace
 - targets
 - profiling, 341
- tracepoints
 - disabling, 104
 - enabling, 104
 - markers, 103
 - overview, 102
 - toggling status of, 104
 - viewing information about, 105
- traffic pane, 50
- Tree Browser, 220
 - calls, browsing, 225 to 226
 - classes, browsing, 224

Index

- node operations, 222
- opening, 220
- rerooting, 223
- Trg** pane, 48
- troubleshooting
 - concurrent freeze-/run-mode connections, 74
- Turn Off Target on Disconnect** menu item, 512
- tv** command line option, 569
- `_tx_thread_created_ptr`
 - OS-specific symbol, 451
- Type** menu item, 519
- types
 - in Data Explorer, 175
 - data, browsing, 193 to 194, 214 to 216
 - shortcut menu for, 550 to 551
- typographical conventions, xviii

U

- Unconnected Executables** target list message, 36
- UNIX
 - file chooser dialog box, 554
 - Print Setup** dialog box, 501
 - printing, 501
- unknown programs, 94
- Unload Module** menu item, 520
- Unload Symbols** menu item, 506
- Unreadable memory Data Explorer message, 181
- update** command, 169
- Upstack** button, 535, 560
- UpStack** menu item, 136, 516
- UpStack To Source** menu item, 137, 516
- usage** command line option, 569
- Use Connection** menu item, 506
- Use Which Connection/CPU?** dialog box, 87
- using a connection, 87
- Utility Program Launcher**, 524

V

- V** command line option, 569
- variable lifetime information, 278
- variables
 - global, listing, 517
 - local, listing, 513, 517
 - in procedure, listing, 518
 - `$result`, 298
 - search designators, 277
 - shortcut menu for, 549
 - static, listing, 517
 - system, 291 to 292
 - viewing value of, 279
 - viewing values of, 277
- Variables In Procedure** menu item, 518
- VERIFYHALT system variable, 292
- verifying presence of executable, 95
- view** command, 164
- view descriptions
 - MULTI data visualization (**.mdv**) files, 574, 602
- View Expression** menu item, 514
- View** menu, 512
- viewdel** command, 164
- viewing
 - assembly code, 41
 - caches, 379, 383
 - Debugger version information, 569
 - information about MULTI, 531
 - interlaced assembly code, 41
 - line numbers, 39, 42
 - local variables, 513
 - memory, 304
 - See also* **Memory View** window
 - native processes, 378
 - profiling data, 347, 350, 363
 - program at addresses, 138, 516
 - source code, 41
 - variables, 277, 279
- viewlist** command, 172
- visualizations

Index

custom, *see* MULTI data visualization

W

wildcards, 284

windows

 closing multiple, 515

 display problems, fixing, 569

 in MULTI, 130

 opening multiple, 32

 reusing, 32

 stand-alone, 144

Windows menu, 530

-wmhints command line option, 569

workspaces

 overview, 5

Write to file menu item, 499

X

X Window managers

 fixing window display problems, 569

Z

ZOMBIE status bar message, 54

Zombied target list status, 37